IST 718:
Big Data Analytics

UNIT 3-1 Hadoop, MapReduce and Yarn

Note: MATERIAL here is from a mix of Readings & sources as listed in References Slide(s)

01 Systems Review

Distributed Systems

CAP Theorem

Hadoop and Yarn

MapReduce

03 Distributed Systems

A distributed system is a collection of independent computers that appear to the users as a single coherent system

Distributed System Advantages (compare to single system):

- It could be cheaper: building a supercomputer is expensive
- Faster processing: parallel processing
- Reliability: if one node fails, then we can process in a different node
- Incremental growth: add more computers as needed

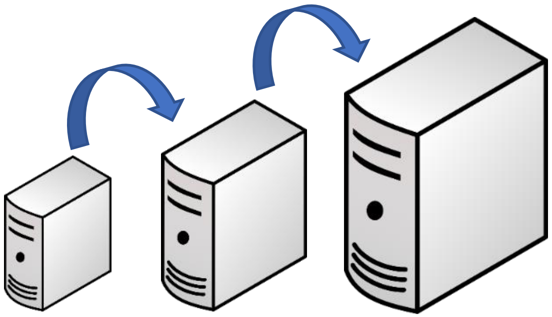
Distributed System Disadvantage:

- Software must be customized
- Network: often this is the bottleneck
- More components to fail
- Complex security

03 Addressing Growth (Scaling)

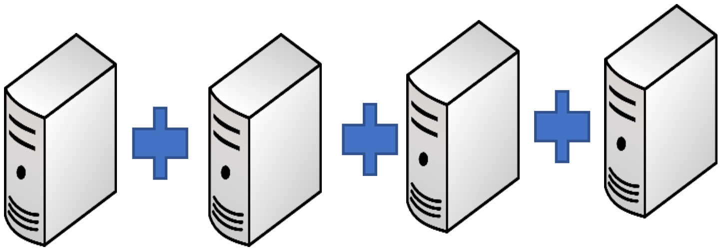
Vertical "Scale Up"

- Add more resources to an existing system running the service.
- Easier, but limited scale.
- Single point of failure



Horizontal "Scale Up"

- Run the service over multiple systems and orchestrate communication between them.
- Harder, but massive scale.
- Overhead to manage nodes.

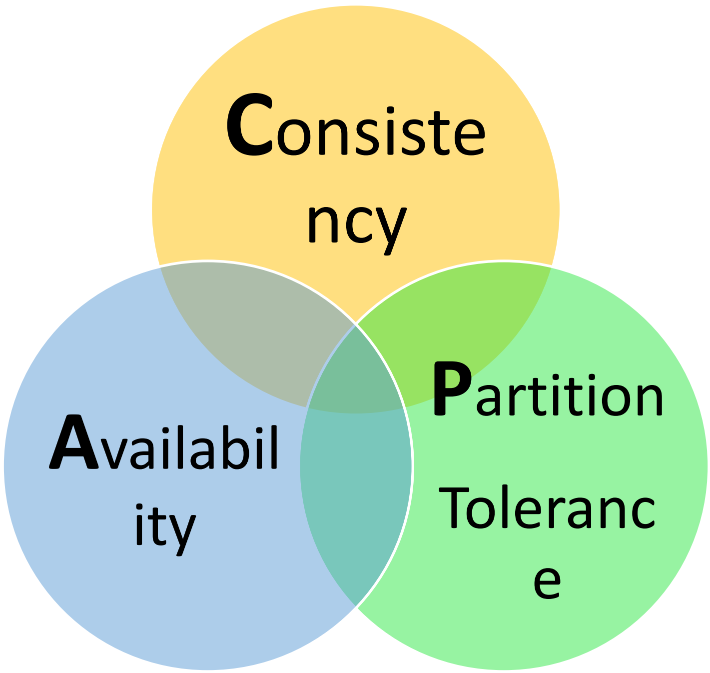


03 CAP Theorem of Distributed Systems

- Also called "Brewer's Theorem" after computer scientist Eric Brewer.
- From a storage perspective, there are many goals for a distributed system.
- Almost all goals of distributed systems can be fit into one of these categories
  - Data Consistency: all nodes see the same data at the same time.
  - Data Availability: assurances that every request can be processed.
  - Partition Tolerance: network failures are tolerated, the system continues to operate

03 CAP Theorem of Distributed Systems

System	Consistency	Availability	Partition Tolerance
Email system		✓	✓
Bank account (debit, credit)	✓	✓	
Databases	✓	✓	
Distributed system for ML	✓		?
Distributed system for log dumping		✓	?



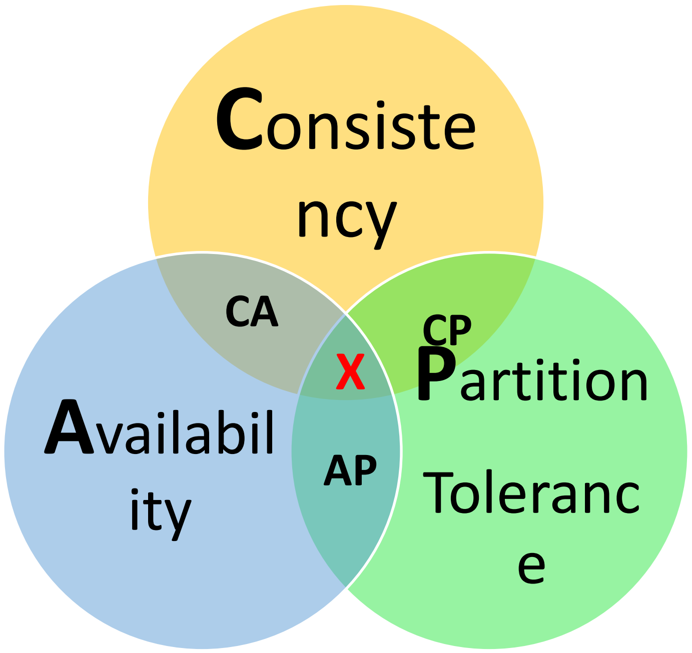
- **Data Consistency**: all nodes see the same data at the same time.
- **Data Availability**: assurances that every request can be processed.
- **Partition Tolerance**: network failures are tolerated, the system continues to operate

CAP Theorem of Distributed Systems

- However, there is a fundamental constraint: only two of the goals can be fulfilled at the same time
  - **Data Consistency**: all nodes see the same data at the same time.
  - **Data Availability**: assurances that every request can be processed.
  - **Partition Tolerance**: network failures are tolerated, the system continues to operate

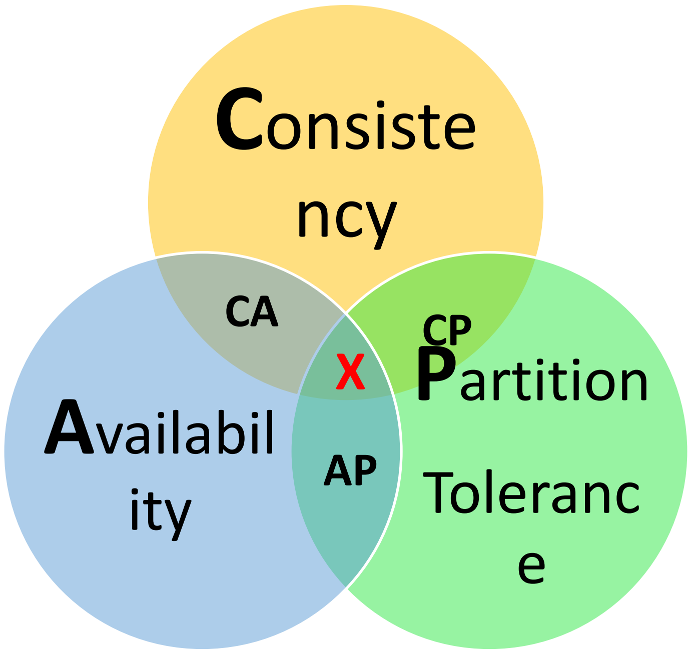
CAP Theorem of Distributed Systems

- However, there is a fundamental constraint: only two of the goals can be fulfilled at the same time
  - **Data Consistency**: all nodes see the same data at the same time.
  - **Data Availability**: assurances that every request can be processed.
  - **Partition Tolerance**: network failures are tolerated, the system continues to operate



CAP Theorem of Distributed Systems

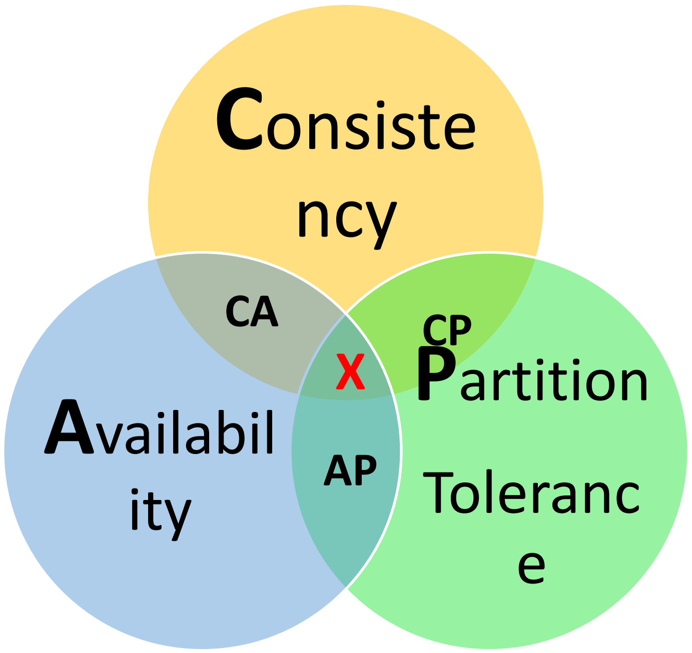
System	Consistency	Availability	Partition Tolerance
Email system			
Bank account (debit, credit)			
Databases			
Distributed system for ML			
Distributed system for log dumping			



Data Consistency: all nodes see the same data at the same time.
Data Availability: assurances that every request can be processed.
Partition Tolerance: network failures are tolerated, the system continues to operate

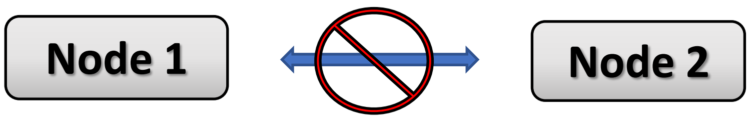
CAP Theorem of Distributed Systems

System	Consistency	Availability	Partition Tolerance
Email system		☑	☑
Bank account (debit, credit)	☑	☑	
Databases	☑	☑	
Distributed system for ML	☑		?
Distributed system for log dumping		☑	?



Data Consistency: all nodes see the same data at the same time.
Data Availability: assurances that every request can be processed.
Partition Tolerance: network failures are tolerated, the system continues to operate

Why Can’t You Have All Three? *



A ***Counterexample***:

- Suppose we lose communication between nodes:
  - We must ignore any updates the nodes receive, or sacrifice **Consistency,S** or we must deny service until it becomes ***Available*** again.
- If we guarantee ***Availability*** of requests, despite the failure:
  - We gain ***Partition Tolerance*** (the system still works), but lose ***Consistency*** (nodes will get out of sync).
- If we guarantee **Consistency** of data, despite the failure:
  - We gain ***Partition Tolerance*** (again, system works) but lose ***Availability*** (data on nodes cannot be changed until failure is resolved).

- Can have all three some of the time
- Can not have all three persistently all the time at the same time.

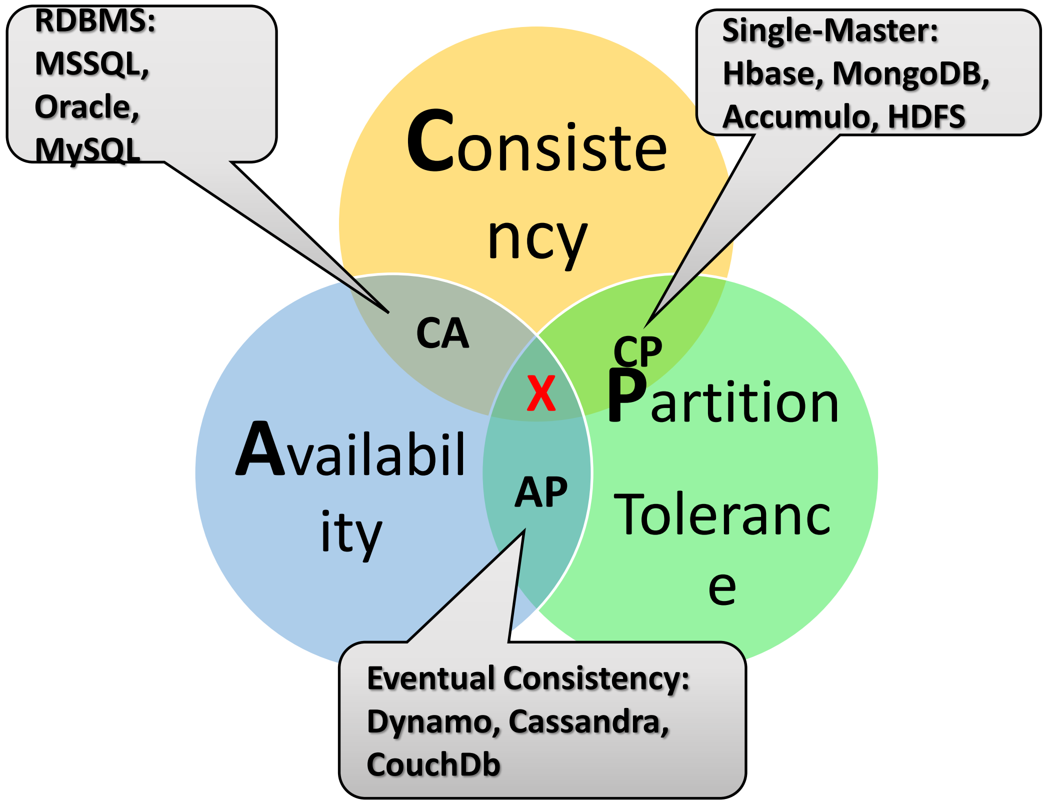
Why Can’t You Have All Three?

- Another way to think about CAP:
  - No network is safe from failure so network partitioning generally has to be tolerated.
  - Thus, in the presence of a network partition, we need to choose consistency or availability.
  - If the network is operating correct, availability and consistency can be satisfied simultaneously.

Why Can't You Have All Three?

- If there is a network partition failure
  - Choosing availability over consistency means we always return a value even if it can't be guaranteed to be the latest value.
  - Choosing consistency over availability means that an error condition or a time out happens if the requested information cannot be guaranteed to be the most recent.

CAP Theorem Database Examples



CAP Theorem Database Examples

- Relational Database Management Systems (RDBM) like Oracle, MySQL and SQL Server:
  - Focus on Consistency and Availability
  - Use ACID Principles (See https://en.wikipedia.org/wiki/ACID) : Atomicity, Consistency, Isolation, Durability
  - Sacrifice Partition Tolerance and thus they don't scale well horizontally.
  - Use cases: Business data, when you don't need to scale out.
- Single-Master systems like MongoDb, Hbase, Redis, and HDFS:
  - Provide Consistency at scale but data availability runs through a single node.
  - Use cases: Read-heavy. Caching, document storage, product catalogs.
- Eventual Consistency systems like CouchDb, Cassandra and Dynamo
  - Provide Availability at scale but do not guarantee consistency.
  - Use cases: Write heavy, Isolated activities: Shopping carts, Orders, Tweets.

What is Hadoop?

- Apache Hadoop includes a storage system called the Hadoop file system (HDFS), and a computing system called MapReduce
- HDFS and MapReduce are closely integrated
- HDFS is designed for low-cost storage over clusters of commodity servers
- Difficult to run MapReduce without Hadoop

What is Mapreduce?

- MapReduce is a programming model and an associated implementation for processing and generating large data sets
- See https://en.wikipedia.org/wiki/MapReduce
- More on mapreduce later in this slide deck

Birthplace of Hadoop:

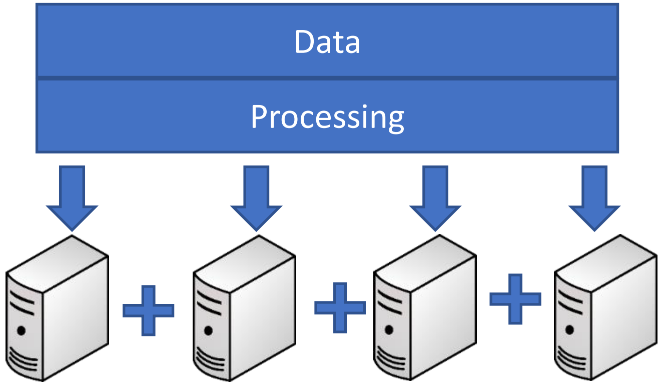
- Google, Facebook, Yahoo!
- These companies had so much data, that **enterprise DBMSs** could not meet their reporting requirements.

Time to process 1 day of data

Number of Hours in a day.

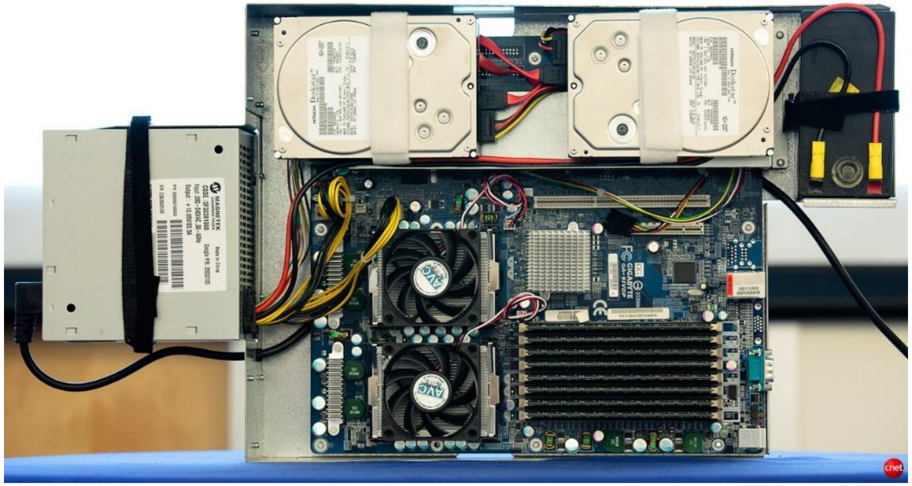
Hadoop Handles Big Data By ***Scaling Out***

- **Problem**: File Too Large to fit on a single storage platform?
- **Solution**: Distribute the file over several computers.
- **Problem**: Server not “fast” enough to process your data.
- **Solution**: Distribute data processing over several computers.



Hadoop is Designed to Use *Commodity Hardware*

- Hadoop Hardware
  - Modular
  - Easy to add and remove nodes.
  - Failure is not only acceptable, but *expected*.
- This is contrary to enterprise hardware
  - High-redundancy / Fault-tolerant
  - Vertical Scaling
  - Storage arrays



* Google Hardware spec for server source: C|NET

How Does Hadoop Store, Process and Manage Big Data?

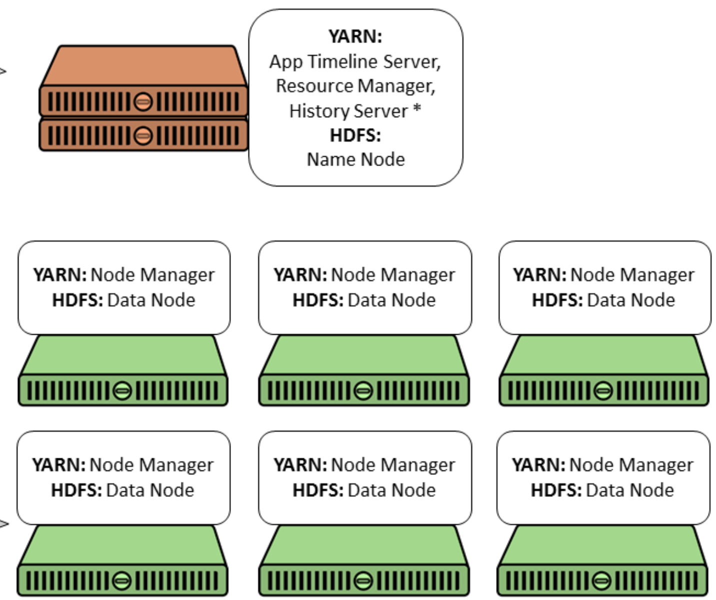
Hadoop Clusters

3 Node Types:

1. Master Nodes
2. Worker Nodes
3. Client Nodes

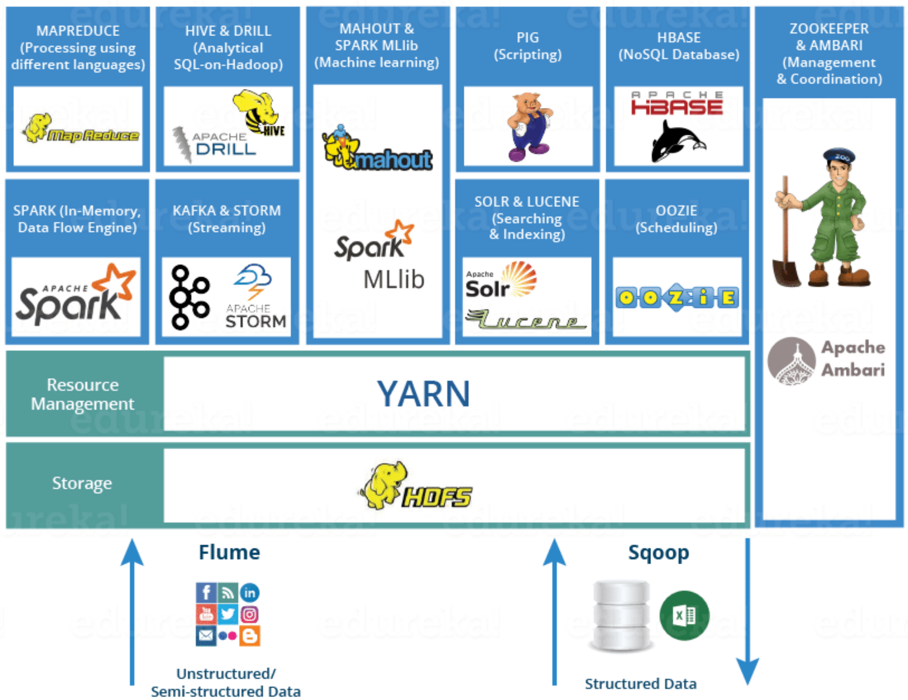
Master Node:
- Manage the Hadoop infrastructure.
- Runs *one* of each of these services per cluster, on a single server or many.
- Should run on server-class hardware.

Worker Nodes:
- Store data and perform processing over it.
- Each node runs the same services.
- Runs on commodity hardware.



* Map Reduce 2 service on YARN

03 Hadoop 2.0 Ecosystem



https://www.edureka.co/blog/hadoop-ecosystem

HDFS: Hadoop Distributed File System

YARN: Yet Another Resource Negotiator

MapReduce: Distributed Processing Framework

Spark: In-memory Data Processing

PIG, HIVE: Data Processing Services using Query (SQL-like)

HBase: NoSQL Database

Mahout, Spark MLlib: Machine Learning

Apache Drill: Schema-free SQL for Hadoop

Zookeeper: Distributed Co-ordination Service

Oozie: Workflow Scheduler System

Flume, Sqoop: Data Ingesting Services

Solr & **Lucene**: Searching & Indexing

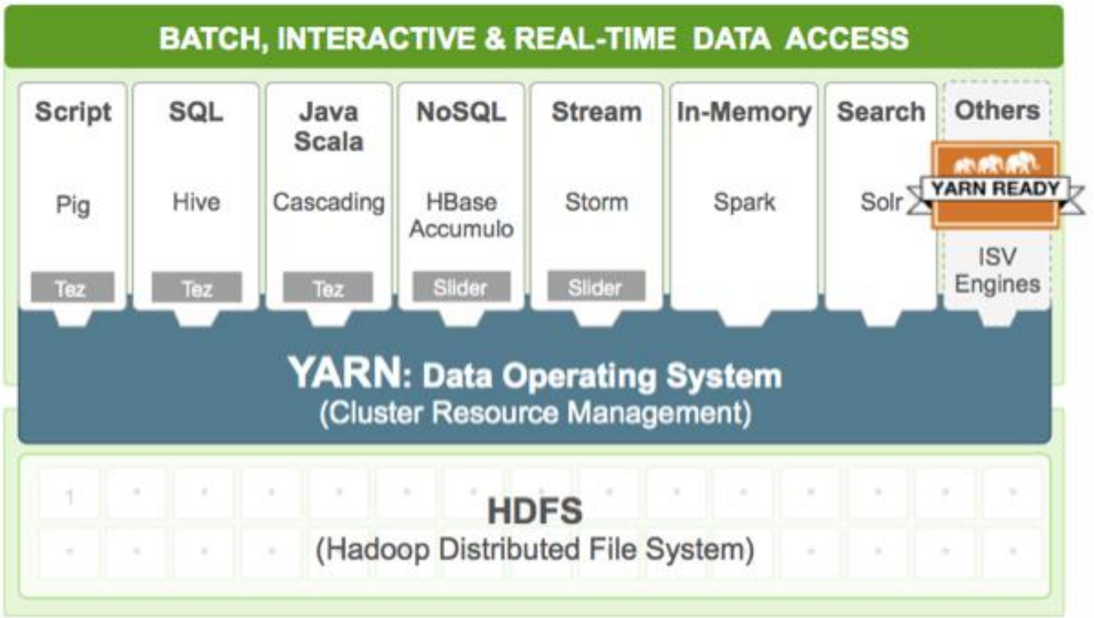
Ambari: Provision, Monitor & Maintain clusters

Historical Anecdote

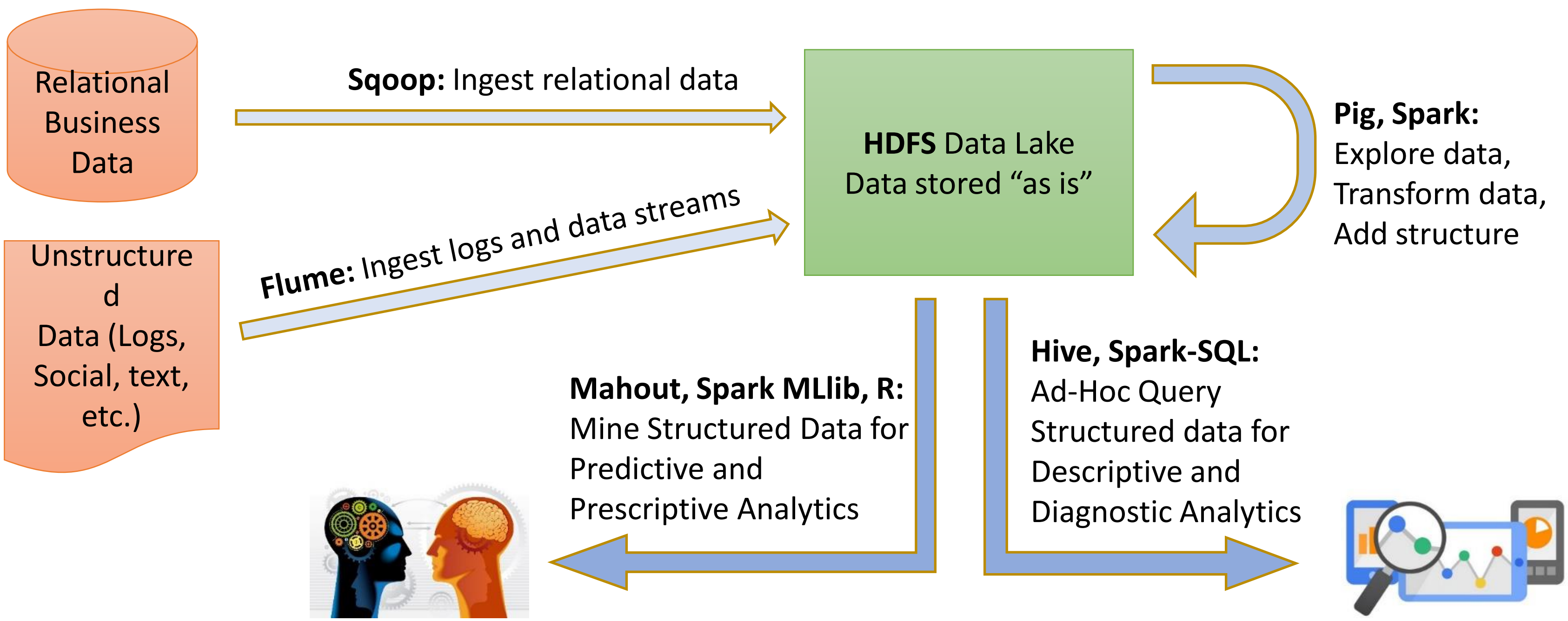
https://www.newyorker.com/magazine/2018/12/10/the-friendship-that-made-google-huge

YARN: The Data Operating System

- Hadoop 2.0 Introduces YARN (Yet Another Resource Negotiator)
- Orchestrates processing over the nodes.
- Uses HDFS for storage.
- Runs a variety of Applications.



An Over-Simplified Version of the Hadoop Ecosystem in Action



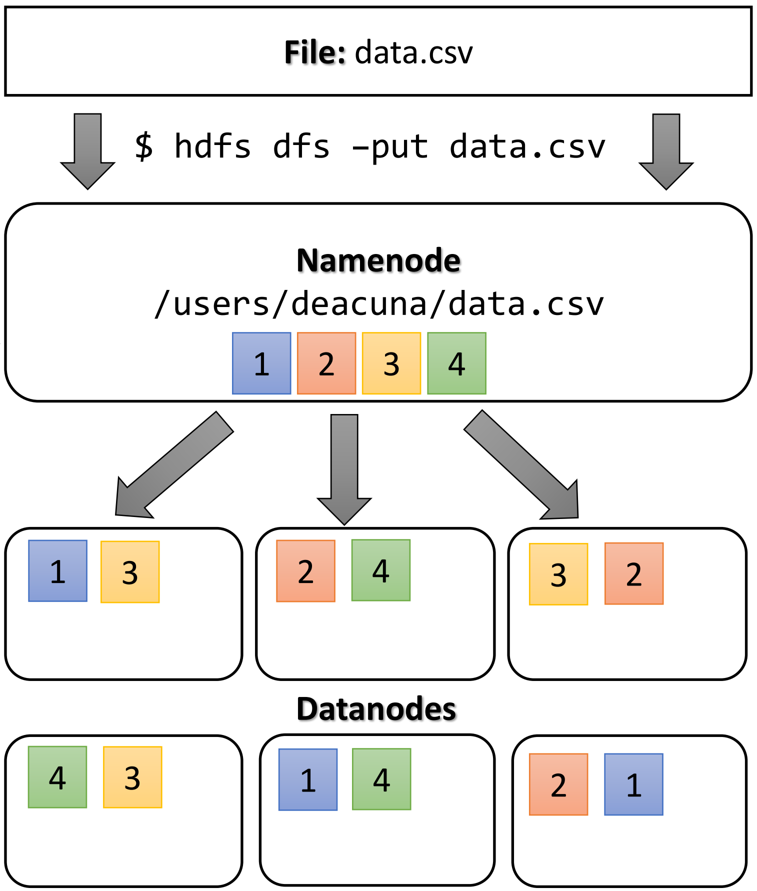
HDFS: Hadoop Distributed File System

- Based on Google’s GFS (Google File System)
- Data Distributed over Physical Nodes
- Designed for Failover
- Data Stored “as is”
- Data Split into Blocks
- Default Replication factor is 3

HDFS
At Work

Client:
1) Issues command to write data.csv file to HDFS

Namenode – responsible managing file data on the cluster:
2) Splits the file into 128MB blocks (size can be changed).
3) Writes each block to a separate Datanode.
4) Replicates each block a number of times
(default is 3).
5) Keeps track of which nodes contain each block in the file.



How do you get data into HDFS?

- By using the “spark context” object in jupyter notebook:
  - Example: sc.textfile(“file_name”) -> loads data from a file and creates a resilient distributed data object stored using HDFS.
- HDFS Command Line: $ hdfs dfs –put file
- WebHDFS – REST API for HDFS commands
- Sqoop- Import / Export with other DBMS’s
- Flume – Import logs, events, social media
- Storm – real-time event proceessing
- MapReduce Job output

HDFS Commands

`hdfs dfs -command [args]`

Here are a few (of the almost 30) HDFS commands:

`-cat`: display file content (uncompressed)

`-text`: just like cat but works on compressed files

`-chgrp,-chmod,-chown`: changes file permissions

`-put,-get,-copyFromLocal,-copyToLocal`: copies files from the local file system to the HDFS and vice versa.

`-ls, -ls -R`: list files/directories

`-mv,-moveFromLocal,-moveToLocal`: moves files

`-stat`: statistical info for any given file (block size, number of blocks, file type, etc.)

NOTE: Replacement for “Hadoop fs”

Examples of HDFS Commands

- `hdfs dfs `**`-mkdir`**` mydata`
- `hdfs dfs `**`-put`**` numbers.txt mydata/`
- `hdfs dfs `**`-ls`**` mydata`

Remember: This takes effect in HDFS (not locally) and uses the identity of the current user

HDFS File Permissions – like Linux permissions

- Files and directories have owners and groups
- r = read
- w = write
- x = permission to access the contents of a directory

```
drwxr-xr-x   - hue      hue           0 2013-08-27 23:00 /user/hue/oozie/workspaces/unmanaged/shell
-rwxr-xr-x   3 hue      hue          77 2013-08-27 23:00 /user/hue/oozie/workspaces/unmanaged/shell/hello.py
drwxr-xr-x   - hue      hue           0 2013-08-27 23:00 /user/hue/oozie/workspaces/unmanaged/sleep
-rwxr-xr-x   3 hue      hue           0 2013-08-27 23:00 /user/hue/oozie/workspaces/unmanaged/sleep/empty
drwxr-xr-x   - hue      hue           0 2013-08-27 23:00 /user/hue/oozie/workspaces/unmanaged/sqoop
-rwxr-xr-x   3 hue      hue        7175 2013-08-27 23:00 /user/hue/oozie/workspaces/unmanaged/sqoop/TT.java
-rwxr-xr-x   3 hue      hue         420 2013-08-27 23:00 /user/hue/oozie/workspaces/unmanaged/sqoop/db.hsqldb.properties
-rwxr-xr-x   3 hue      hue         276 2013-08-27 23:00 /user/hue/oozie/workspaces/unmanaged/sqoop/db.hsqldb.script
drwxr-xr-x   - hue      hue           0 2013-08-27 23:00 /user/hue/oozie/workspaces/unmanaged/ssh
-rwxr-xr-x   3 hue      hue           0 2013-08-27 23:00 /user/hue/oozie/workspaces/unmanaged/ssh/empty
drwxr-xr-x   - root     root          0 2013-08-29 03:22 /user/root
drwxr-xr-x   - root     root          0 2013-08-29 03:23 /user/root/mydata
-rw-r--r--   3 root     root       2549 2013-08-29 03:23 /user/root/mydata/numbers.txt
-rw-r--r--   3 root     root    3613198 2013-08-28 21:55 /user/root/stocks.csv
[root@sandbox demos]#
```

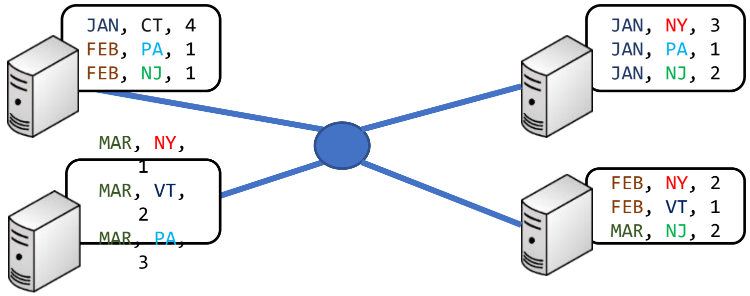
MapReduce

The problem

- We want to use our distributed system to process arbitrarily large amounts of data
- We can develop custom scripts for communicating among computers and solving a problem
- The problem is that this script can be different for each task
- "MapReduce is a programming model and an associated implementation for processing and generating large data sets"
  J. Dean, S. Ghemawat, "*MapReduce: Simplified Data Processing on Large Clusters*", OSDI 2004

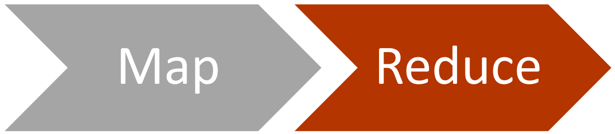
The solution

- Create a simple programming model where there is **no communication** between tasks, enhancing fault tolerance and lowering complexity
- An example application:
  - Very large and distributed dataset of orders per month and state
  - We want to compute total number of orders per month
  - Any thoughts?



MapReduce (programming model)

- Computation takes a set of *input* key/value pairs and produces a set of *output* key/value pairs
- The user implements two functions: *Map* and *Reduce*
- **Map**: takes an input *element* and produces a set of *intermediate* key/value pairs
- **Reduce**: accepts an *intermediate* key and a pair of values for the same key and combines them into zero or one value.
- The program stops when no more key-value pairs can be reduced



MapReduce: example

- **Map**: takes an input *element* and produces a set of *intermediate* key/value pairs

input

$\boxed{1}$ ← **input element**
2
3
4
5
6
7
8
9
1
0

MapReduce: example

- **Map**: takes an input *element* and produces a set of *intermediate* key/value pairs

input

Map

1 ← **input element** → lambda e: (1, e)
2
3
4
5
6
7
8
9
1
0

MapReduce: example

- **Map**: takes an input *element* and produces a set of *intermediate* key/value pairs

input

Map

Intermediate pairs

input		Map	Intermediate pairs
$\boxed{1}$	← **input element** →	lambda e: (1, e)	**(1, 1)**
2			
3			
4			
5			
6			
7			
8			
9			
1			
0			

MapReduce: example

- **Map**: takes an input *element* and produces a set of *intermediate* key/value pairs

input

Map

lambda e: (1, e)

Intermediate pairs

(1, 1)

1
$\boxed{2}$ ← **input element**
3
4
5
6
7
8
9
1
0

MapReduce: example

- **Map**: takes an input *element* and produces a set of *intermediate* key/value pairs

input

Map

lambda e: (1, e)

Intermediate pairs

1
2 ← **input element**
3
4
5
6
7
8
9
1
0

(1, 1)
(1, 2)

MapReduce: example

- **Map**: takes an input *element* and produces a set of *intermediate* key/value pairs

input

Map

lambda e: (1, e)

Intermediate pairs

input		Intermediate pairs
1		(1, 1)
2		(1, 2)
3		(1, 3)
4		(1, 4)
5		(1, 5)
6		(1, 6)
7		(1, 7)
8		(1, 8)
9		(1, 9)
$\boxed{1}$	**input element**	**(1, 10)**
0		

MapReduce: example

- **Map**: takes an input *element* and produces a set of *intermediate* key/value pairs

input		**Map**	*Intermediate pairs*
1		lambda e: (1, e)	(1, 1)
2			(1, 2)
3			(1, 3)
4			(1, 4)
5			(1, 5)
6			(1, 6)
7			(1, 7)
8			(1, 8)
9			(1, 9)
$\boxed{1}$	**input element**		**(1, 10)**
0			

Identify 1) input, 2) input element, 3) map function (3.1 rule for key, 3.2 rule for value), 6) intermediate key / value

MapReduce: example

- **Reduce**: accepts an *intermediate* key and a **pair of values** for the same key and combines them into zero or one value.

Intermediate pairs

(1, 1)
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
(1, 7)
(1, 8)
(1, 9)
(1, 10)

pair values (same key)

MapReduce: example

- **Reduce**: accepts an *intermediate* key and a **pair of values** for the same key and combines them into zero or one value.

Intermediate pairs

(1, $\boxed{1}$)
(1, 2)
(1, $\boxed{3}$)
(1, 4)
(1, 5)
(1, 6)
(1, 7)
(1, 8)
(1, 9)
(1, 10)

pair values (same key)

reduce

lambda v1, v2: v1 + v2

MapReduce: example

- **Reduce**: accepts an *intermediate* key and a **pair of values** for the same key and combines them into zero or one value.

Intermediate pairs

(1, 1)
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
(1, 7)
(1, 8)
(1, 9)
(1, 10)

pair values (same key)

reduce

lambda v1, v2: v1 + v2

Intermediate pairs

(1, 4)

MapReduce: example

- **Reduce**: accepts an *intermediate* key and a **pair of values** for the same key and combines them into zero or one value.

Intermediate pairs			***Intermediate pairs***
(1, 1)		**reduce**	**(1, 4)**
(1, 2)	**pair values (same key)**	lambda v1, v2: v1 + v2	(1, 2)
(1, 3)			
(1, 4)			(1, 4)
(1, 5)			(1, 5)
(1, 6)			(1, 6)
(1, 7)			(1, 7)
(1, 8)			(1, 8)
(1, 9)			(1, 9)
(1, 10)			(1, 10)

MapReduce: example

- **Reduce**: accepts an *intermediate* key and a **pair of values** for the same key and combines them into zero or one value.

Intermediate pairs

(1, 1)
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
(1, 7)
(1, 8)
(1, 9)
(1, 10)

pair values (same key)

reduce

lambda v1, v2: v1 + v2

Intermediate pairs

(1, 4)
(1, 2)

(1, 4)
(1, 5)
(1, 6)
(1, 7)
(1, 8)
(1, 9)
(1, 10)

Outputs are always going to be sorted by key

MapReduce: example

- **(Repeat) Reduce**: accepts an *intermediate* key and a **pair of values** for the same key and combines them into zero or one value.

Intermediate pairs **(*t*+1)**	***Intermediate pairs*** **(*t*)**
(1, 4)	**(1, 4)**
(1, 2)	(1, 2)
(1, 4)	(1, 4)
(1, 5)	(1, 5)
(1, 6)	(1, 6)
(1, 7)	(1, 7)
(1, 8)	(1, 8)
(1, 9)	(1, 9)
(1, 10)	(1, 10)

Next iteration

MapReduce: example

- **(Repeat) Reduce**: accepts an *intermediate* key and a **pair of values** for the same key and combines them into zero or one value.

Intermediate pairs

(1, 4)
(1, 2)

(1, 4)
(1, 5)
(1, 6)
(1, 7)
(1, 8)
(1, 9)
(1, 10)

pair values (same key)

reduce

lambda v1, v2: v1 + v2

Intermediate pairs

MapReduce: example

- **(Repeat) Reduce**: accepts an *intermediate* key and a **pair of values** for the same key and combines them into zero or one value.

Intermediate pairs

(1, 4)
(1, 2)

(1, 4)
(1, 5)
(1, 6)
(1, 7)
(1, 8)
(1, 9)
(1, 10)

pair values (same key)

reduce

lambda v1, v2: v1 + v2

Intermediate pairs

(1, 8)
(1, 2)

(1, 5)
(1, 6)
(1, 7)
(1, 8)
(1, 9)
(1, 10)

MapReduce: example

- **(Repeat) Reduce**: accepts an *intermediate* key and a **pair of values** for the same key and combines them into zero or one value.

Intermediate pairs

(1, 8)
(1, 2)

(1, 5)
(1, 6)
(1, 7)
(1, 8)
(1, 9)
(1,
10)

pair values (same key)

reduce

lambda v1, v2: v1 + v2

Intermediate pairs

**(1,
13)**
(1, 2)
(1, 6)
(1, 7)
(1, 8)
(1, 9)
(1,
10)

Identify 1) intermediate key, 2) pairs of values, 3) reduce function, 4) output value

MapReduce: example

- **(Repeat) Reduce**: accepts an *intermediate* key and a **pair of values** for the same key and combines them into zero or one value.

Intermediate pairs

(1, $\boxed{8}$)
(1, 2)

pair values (same key)

(1, $\boxed{5}$)
(1, 6)
(1, 7)
(1, 8)
(1, 9)
(1, 10)

reduce

lambda v1, v2: v1 + v2

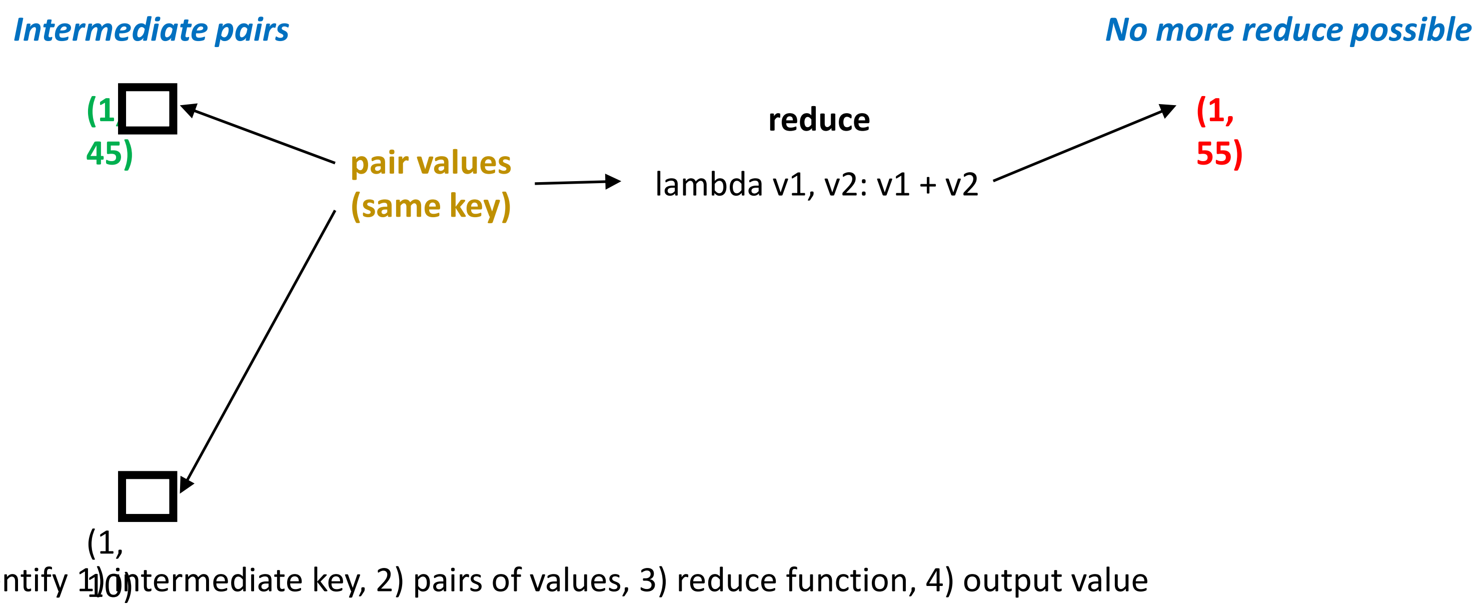
Intermediate pairs

(1, 13)
(1, 2)
(1, 6)
(1, 7)
(1, 8)
(1, 9)
(1, 10)

Identify 1) intermediate key, 2) pairs of values, 3) reduce function, 4) output value

MapReduce: example

- **(Repeat) Reduce**: accepts an *intermediate* key and a **pair of values** for the same key and combines them into zero or one value.



Identify 1) intermediate key, 2) pairs of values, 3) reduce function, 4) output value

Example Mapreduce Code

Some of the examples in the following slides can be found at the end of the Introduction to Spark and DataFrames notebook

MapReduce: example

- **Map**: takes an input *element* and produces a set of *intermediate* key/value pairs
- **Reduce**: accepts an *intermediate* key and a **pair of values** for the same key and combines them into zero or one value.

input		*Intermediate pairs*		*Output*
1		(1, 1)		
2		(1, 2)		
3		(1, 3)		
4	**map**	(1, 4)	**reduce**	**(1, 55)**
5	lambda e: (1, e)	(1, 5)	lambda v1, v2: v1 + v2	
6		(1, 6)		
7		(1, 7)		
8		(1, 8)		
9		(1, 9)		
10		(1, 10)		

<u>What is the goal of this MR job?</u>

MapReduce: example

- In MapReduce jobs the *map* and *reduce* order of application does not matter

input		*Intermediate pairs*		*Output*
1		(1, 5)		
2		(1, 2)		
3		(1, 4)		**?**
4	**map**	(1, 6)	**reduce**	
5	lambda e: (1, e)	(1, 10)	lambda v1, v2: v1 + v2	
6		(1, 1)		
7		(1, 7)		
8		(1, 8)		
9		(1, 9)		
1		(1, 3)		
0				

MapReduce: example

- **Map**: takes an input *element* and produces a set of *intermediate* key/value pairs
- **Reduce**: accepts an *intermediate* key and a **pair of values** for the same key and combines them into zero or one value.

input

1
2
3
4
5
6
7
8
9
1
0

map

lambda e: (1, e)

reduce

lambda v1, v2: max(v1, v2)

<u>What is the goal of this MR job?</u>

MapReduce: example

- **Map**: takes an input *element* and produces a set of *intermediate* key/value pairs
- **Reduce**: accepts an *intermediate* key and a **pair of values** for the same key and combines them into zero or one value.

input

1
2
3
4
5
6
7
8
9
1
0

map

lambda e: (1, e)

reduce

lambda v1, v2: min(v1, v2)

<u>What is the goal of this MR job?</u>

MapReduce: example

- Map can generate several keys

Input (text)

"Document one"
"Another document"
"A document"

map

```
def f(e):
   results = []
   for word in e.split():
      results.append((word, 1))
   return results
```

reduce

```
lambda v1, v2: v1 + v2
```

<u>What is the goal of this MR job?</u>

MapReduce: worked out example

- Given the following set of records of Month, State, and orders
  - Compute the total number of orders per month using map reduce

```
JAN, NY, 3
JAN, PA, 1
JAN, NJ, 2
JAN, CT, 4
FEB, PA, 1
FEB, NJ, 1
FEB, NY, 2
FEB, VT, 1
MAR, NJ, 2
MAR, NY, 1
MAR, VT, 2
MAR, PA, 3
```

MapReduce: worked out example (2)

- Compute the total number of orders per month using map reduce

Data

```
JAN, NY, 3
JAN, PA, 1
JAN, NJ, 2
JAN, CT, 4
FEB, PA, 1
FEB, NJ, 1
FEB, NY, 2
FEB, VT, 1
MAR, NJ, 2
MAR, NY, 1
MAR, VT, 2
MAR, PA, 3
```

Map([JAN, NY, 3]) → [key_1, $value_1$]

Map([JAN, NJ, 2]) → [key_2, $value_2$]

Map([FEB, PA, 1]) → [key_3, $value_3$]

MapReduce: worked out example (3)

- Compute the total number of orders per month using map reduce

Data

```
JAN, NY, 3
JAN, PA, 1
JAN, NJ, 2
JAN, CT, 4
FEB, PA, 1
FEB, NJ, 1
FEB, NY, 2
FEB, VT, 1
MAR, NJ, 2
MAR, NY, 1
MAR, VT, 2
MAR, PA, 3
```

[key_1, $value_1$]

[key_2, $value_2$]

[key_3, $value_3$]

[key_4, $value_4$]

if $key_1 == key_2$

reduce($value_1$, $value_2$)

MapReduce: worked out example (3)

- Compute the total number of orders per month using map reduce

Data

```
JAN, NY, 3
JAN, PA, 1
JAN, NJ, 2
JAN, CT, 4
FEB, PA, 1
FEB, NJ, 1
FEB, NY, 2
FEB, VT, 1
MAR, NJ, 2
MAR, NY, 1
MAR, VT, 2
MAR, PA, 3
```

[key_1, $value_1$]

[key_2, $value_2$]

[key_3, $value_3$]

[key_4, $value_4$]

if $key_1 == key_2$

reduce($value_1$, $value_2$) → [key_1, $value_{new}$]

if $key_1 == key_3$

reduce($value_{new}$, $value_3$)

MapReduce: worked out example (3)

Data

```
JAN, NY, 3
JAN, PA, 1
JAN, NJ, 2
JAN, CT, 4
FEB, PA, 1
FEB, NJ, 1
FEB, NY, 2
FEB, VT, 1
MAR, NJ, 2
MAR, NY, 1
MAR, VT, 2
MAR, PA, 3
```

[key_1, $value_1$]

[key_2, $value_2$]

[key_3, $value_3$]

[key_4, $value_4$]

if $key_1 == key_2$
reduce($value_1$, $value_2$) → [key_1, $value_{new}$]

if $key_1 == key_3$
reduce($value_{new}$, $value_3$)

Map([JAN, NY, 3]) → [JAN, 3]

Map([JAN, PA, 1]) → [JAN, 1]

Reduce(3, 1)
(Implicitly key=JAN) → 4

MapReduce: worked out example (2)

- Given the following set of records of Month, State, and orders
  - Compute the total number of orders per month using map reduce

```
def map(datapoint):
    ??

def reduce(value1, value2):
    ??
```

```
JAN, NY, 3
JAN, PA, 1
JAN, NJ, 2
JAN, CT, 4
FEB, PA, 1
FEB, NJ, 1
FEB, NY, 2
FEB, VT, 1
MAR, NJ, 2
MAR, NY, 1
MAR, VT, 2
MAR, PA, 3
```

MapReduce: worked out example (2)

- Given the following set of records of Month, State, and orders
  - Compute the total number of orders per month using map reduce

```
def map_func(datapoint):
    return [datapoint[0], datapoint[2]]

def reduce_func(value1, value2):
    return value1 + value2
```

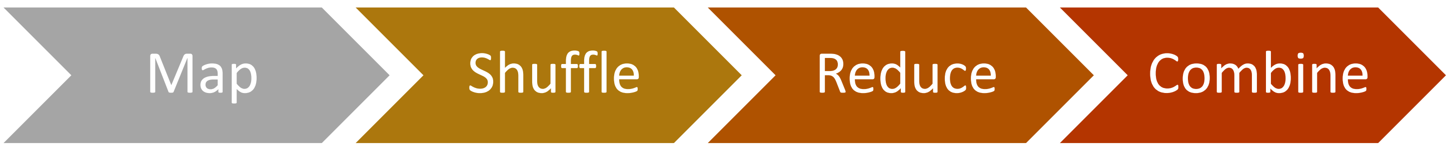
```
JAN, NY, 3
JAN, PA, 1
JAN, NJ, 2
JAN, CT, 4
FEB, PA, 1
FEB, NJ, 1
FEB, NY, 2
FEB, VT, 1
MAR, NJ, 2
MAR, NY, 1
MAR, VT, 2
MAR, PA, 3
```

MapReduce: theory

- It uses properties of functional programming
- Operations do not change data structures
- Original data always exists unmodified
- Data flows are implicitly defined by program design
- Order of operation does not matter
- This means:
  - Easy to parallelize
  - Fault-tolerant: re-execute failed operation
  - Status and monitoring
  - Easy abstraction

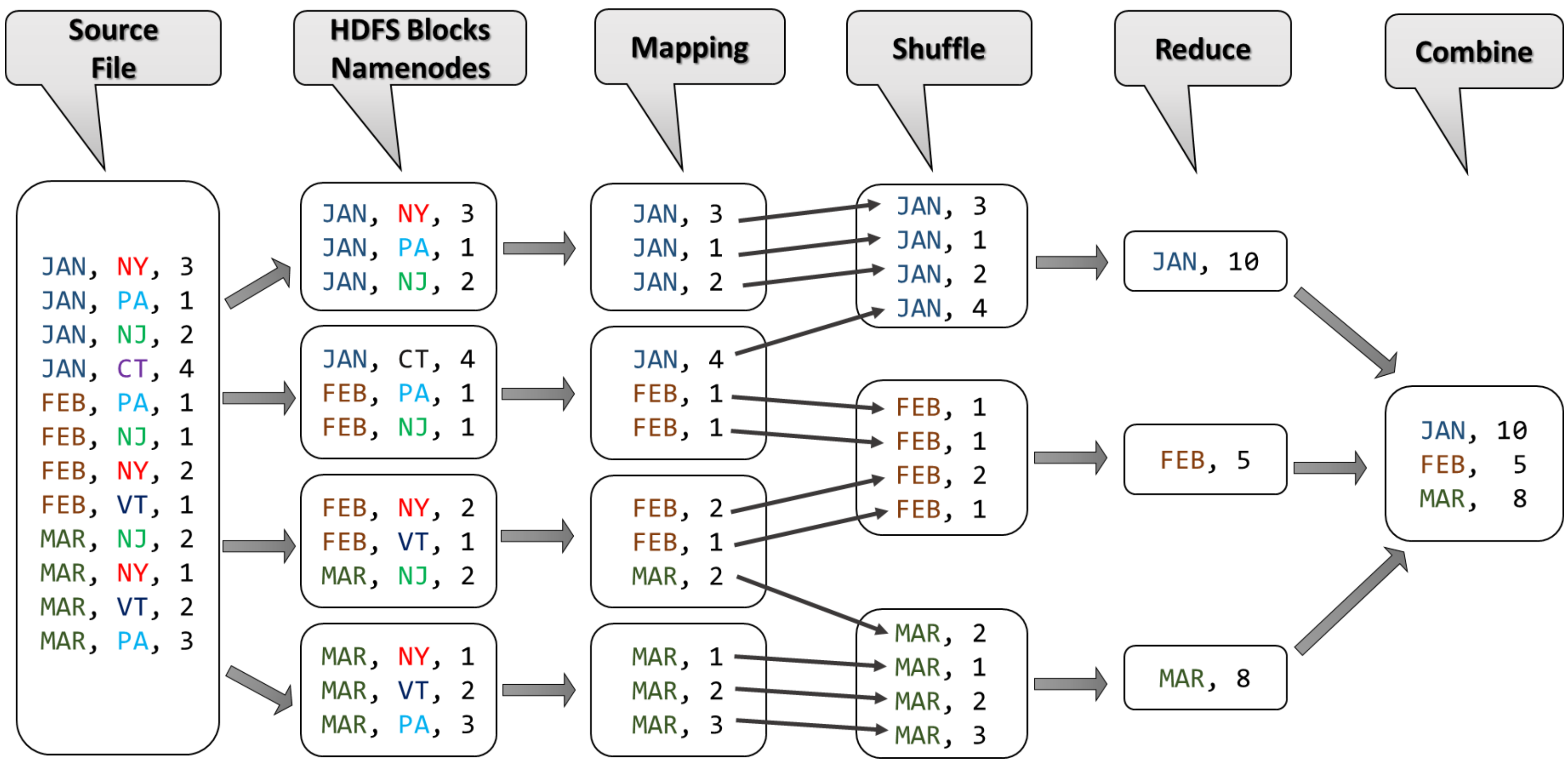
MapReduce (actual steps)

- The actual implementation of the process relies on four steps



- **Shuffle** and **combine** are largely *transparent to the user*
  - Shuffle → Transfer output from mapper to reducer nodes with similar keys
  - Combine → Output of reducer nodes into single output.
- In Hadoop 2.0, MapReduce programs use HDFS and YARN.
- MapReduce is also implemented in Spark

MapReduce Example: Orders for each Month

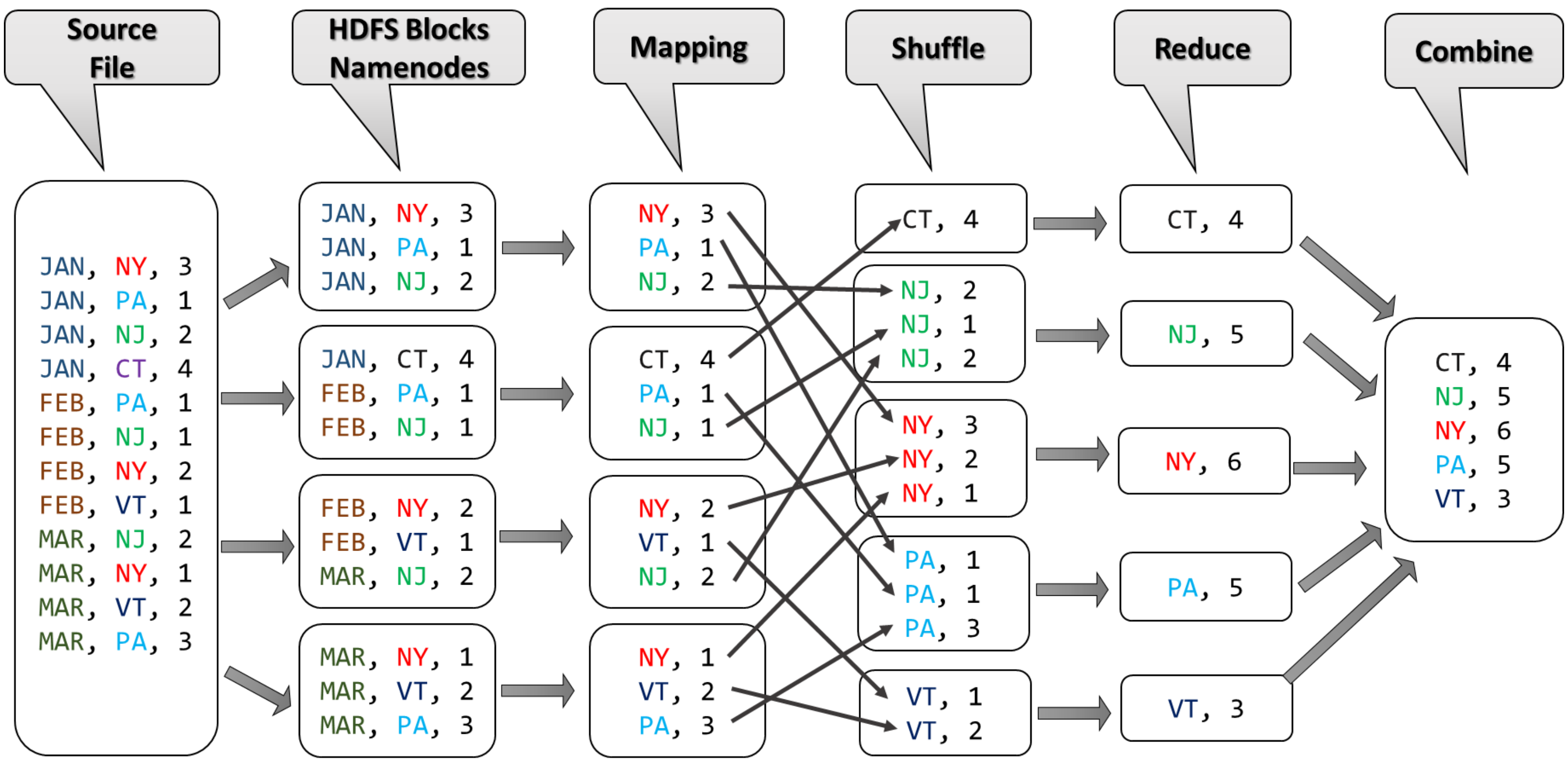


MapReduce: worked out example (2)

- Given the following set of records of Month, State, and orders
  - Compute the total number of orders per state using map reduce

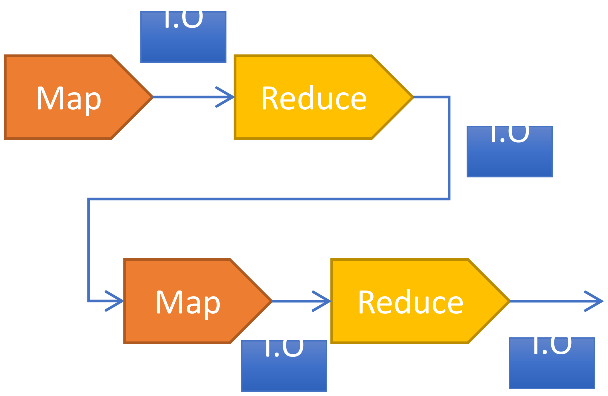
```
JAN, NY, 3
JAN, PA, 1
JAN, NJ, 2
JAN, CT, 4
FEB, PA, 1
FEB, NJ, 1
FEB, NY, 2
FEB, VT, 1
MAR, NJ, 2
MAR, NY, 1
MAR, VT, 2
MAR, PA, 3
```

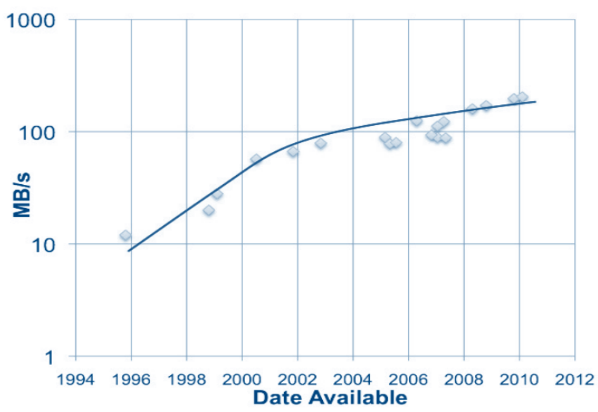
MapReduce: Total Orders by State



Hadoop

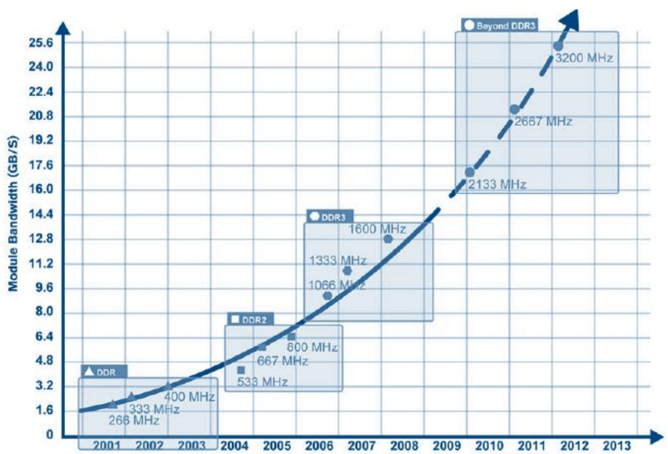
- Traditionally:
  - Hadoop uses single programming model: MapReduce
  - It only works with data on hard drives





Spark

- RAM bandwidth has been increasing exponentially
- Spark can perform in-memory computations



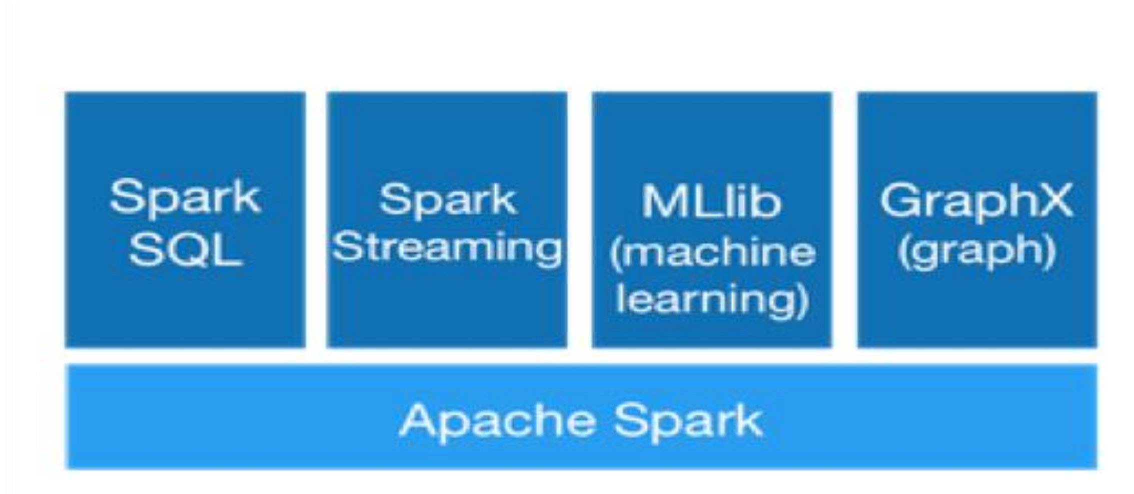
Bandwidths shown for 64-bit memory module. Date indicates approximate industry product introduction.

What is Spark?

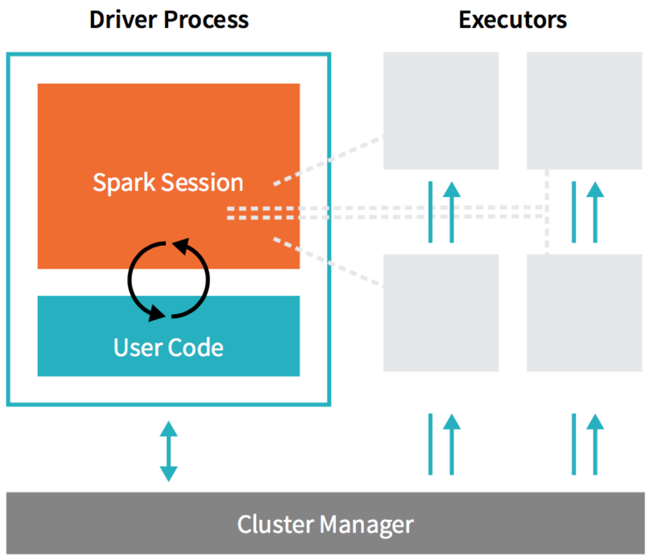
- ***Apache Spark*** is a fast, in-memory analytics system
- Spark has several high-level tools, including:
  - **ML**: a machine learning library
  - **Spark Streaming**: enables high-throughput, fault-tolerant stream processing of live data streams
  - **Spark SQL**: runs SQL and HiveQL queries
  - **GraphX**: an API for graphs and graph-parallel computation
- Spark can be executed in two ways:
  - Standalone
  - With a cluster manager such as YARN

Spark

"Apache Spark™ is a fast and general engine for large-scale data processing."



Spark Architectural Components



- Spark applications consist of a driver process and a set of executor processes.

Spark Driver Process

- The driver process runs the application’s “main” program
- Runs on one of the nodes in the cluster
- Is responsible for three things:
  - Maintaining information about the Spark applications
  - Responding to a user’s program or input
  - Analyzing, distributing, and scheduling work across the executors

Spark Executor Process

- Responsible for actually carrying out the work that the driver assigns to them.
- Each executor is responsible for two things:
  - Executing code assigned to it by the driver
  - Reporting the state of the computation on that executor back to the driver node

Cluster Manager

- Controls the physical machines
- Allocates resources to Spark applications
- Three Cluster Managers
  - Standalone
  - YARN
  - Mesos

Spark 1.6+ vs 2.0+

- Spark 1.6+ relied on transformations on arbitrary datasets known as Resilient Distributed Datasets (RDDs)
- Spark 2.0+ defines more structure in the form of DataFrames, which are similar to tables in SQL and data.frames in R
- The newest versions of Spark define DataSets which are statically-typed DataFrames (more structure)
- With more flexiblity comes greater power but less performance
- Future Spark machine learning will work with DataFrames and not RDD.

Classic Spark v. <2.0

The SparkContext

- The **SparkContext** object performs the following tasks:
  - It connects to the ResourceManager (like YARN) and asks for resources on the Hadoop cluster,
  - Starts executors on the worker nodes in the cluster that the ResourceManager allocated for the Spark application,
  - Sends the application code to the executors,
  - And finally, it sends tasks for the executors to run
- **SparkContext** is represented by the **sc** objec

Spark RDDs

- An RDD (resilient distributed dataset) is a fault-tolerant collection of elements on which operations can be performed in parallel.
- An RDD is an immutable collection that represents:
  - A dataset…
  - …broken up into a list of partitions
  - A list of dependencies on other RDDs
  - An optional list of preferred block locations for an HDFS file
  - Read-only

Important RDD Concepts

- Lineage
  - Information about how an RDD is derived from other datasets or other RDDs
  - RDD is not necessarily materialized all the time due to lazy execution
  - Lineage captured on disk as “lineage graph”
- Persistence
  - Indicate which RDDs which need to keep in memory for reuse
  - User can call “persist” method
- Partitioning
  - RDD elements can be partitioned across machines based on a key in each record

Sample RDD Lineage Graph

- r20 depends on many other RDDs
- See: https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-rdd-lineage.html

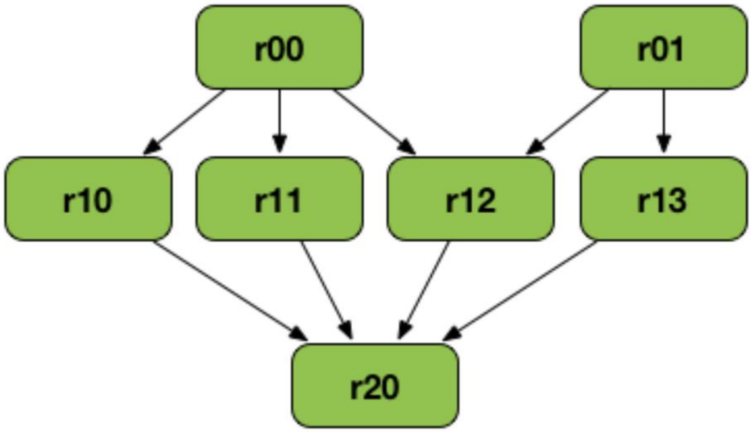


Figure 1. RDD lineage

Creating RDDs

- Can create an initial RDD by applying a transformation to data on disk
- Can create an initial RDD from a code object
- Example ways to create an RDD in Spark:
  - Use the parallelize operation to convert an existing code object into an RDD
  - Use textFile operation to convert a text file on HDFS into an RDD
  - Use sequenceFile operation to convert a binary file on HDFS into an RDD

Example: Creating RDDs

```
from pyspark import SparkContext
```

```
myarray = range(1,20)
myarray
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

```
dist_array = sc.parallelize(myarray)
dist_array
```

```
ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:223
```

RDD Persist

- Spark’s persist method
  - Indicate which RDDs to reuse
  - Indicate if you want to replicate across machines
  - Indicate priority of which in-memory data to spill to disk first
- Example

```
logsrdd = sc.textFile(“hdfs://user/mark/logdata”)
fatals = filter(lambda s: s.startswith(‘FATAL’), logsrdd)
fatals.cache()
fatals.count()
# Notes: logsrdd NOT loaded into RAM because of lazy evaluation
# fatals.cache() = rdd.persist(storageLevel.MEMORY) tries to persist fatals rdd in memory
```

RDD Disk and Recover

- RDD can spill to disk
  - Degrade gracefully (to mapreduce performance)
  - Partitions not in use/lesser use and/or low priority spilled first
- Failure Recovery
  - Recovery is facilitated through redundant RDD block copies across cluster computers.
  - An RDD partition that fails is recovered by the Yarn/Spark Driver
  - Executer applies lineage to prior RDD (or original data on disk) to recover the RDD.

RDD Checkpointing

- Writes an RDD to disk to save the RDD in a specific state
- After checkpointing, future references the RDD don't need to perform upstream transformations in the lineage.
- Similar to caching except that the RDD is stored on disk instead of in memory.
- Example:

```
spark.sparkContext.setCheckpointDir("/some/path/
for/checkpointing")
words.checkpoint()
```

RDD Operations

- There are two types of operations that can be done on RDDs:
  - **Transformations**: create a new dataset/RDD from an existing one
  - **Actions**: Trigger a transformation and instructs spark to compute a result from a series of transformations.
- Transformations
  - lazy - they do not compute their results right away
- Actions
  - Instantiates the RDD
- See https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations for a list of transformations and actions.

RDD Dependencies and Transformations

- Dependency / transformation used synonymously in Spark the Definitive Guide text book
- Two Main Types of Dependencies / Transformation
  - Narrow – child partition depends on only one parent partition
    - E.g. map, filter, union
  - Wide – multiple child partitions depend on one parent partition
    - E.g. Join and group-by transformations
    - Materialize intermediate calculations on parent for fault recovery

Narrow Transformation / Dependency

- Each input partition contributes to one output partition
- See Spark the Definitive Guide, Figure 2.4
- Also called pipelining, spark tries to perform narrow operations in memory

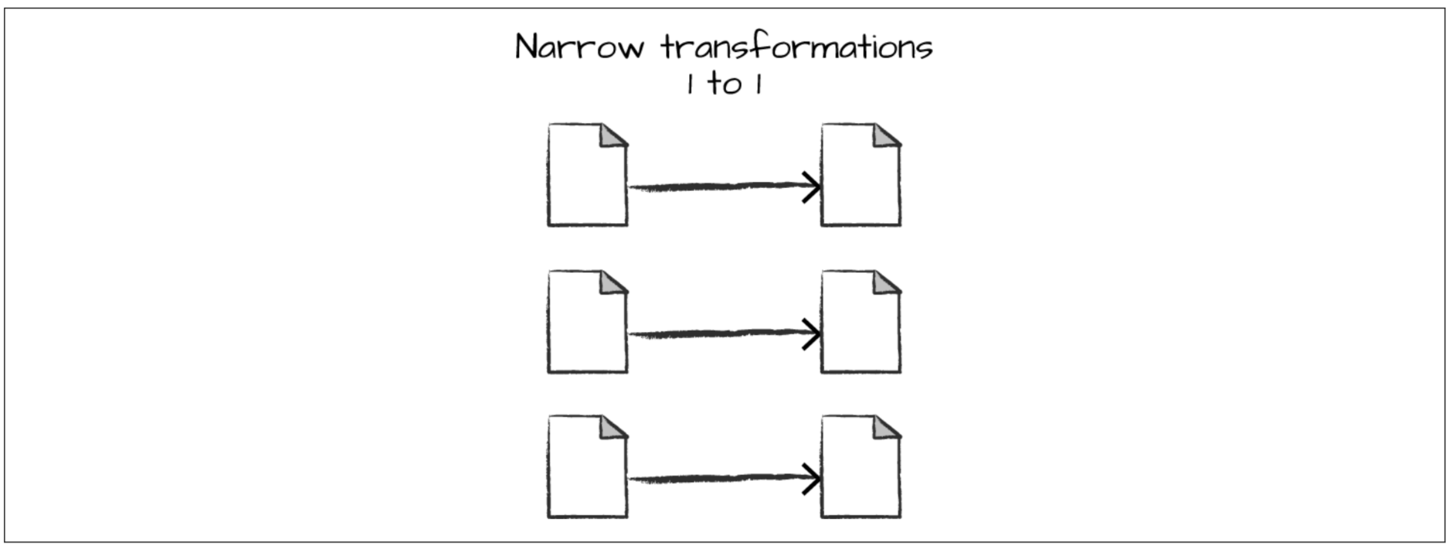


Figure 2-4. A narrow dependency

Wide Transformation / Dependency

- An input partition contributes to many output partitions
- See Spark the Definitive Guide, Figure 2.5
- Also called a shuffle where Spark exchanges partitions across the cluster

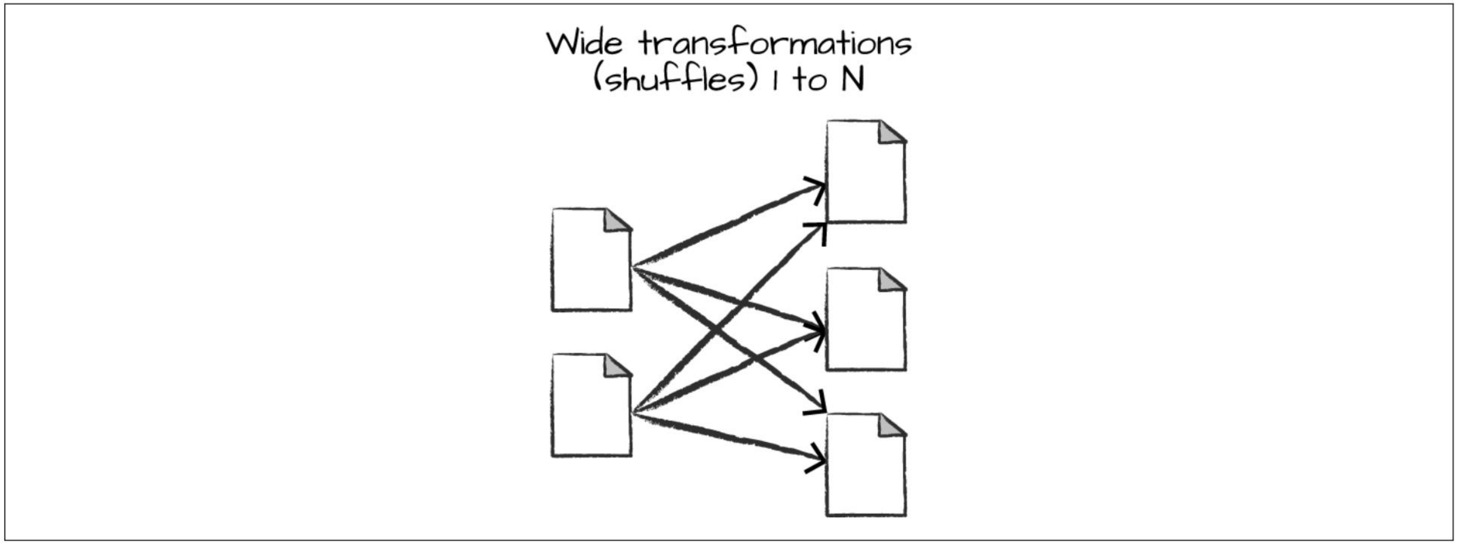


Figure 2-5. A wide dependency

Example Transformations

- **map(*func*):** returns a new distributed dataset formed by passing each element of the source through the function *func*.
- **flatMap(*func*):** same as **map** but when multiple key-value pairs are returned
- **filter(*func*):** return a new dataset formed by selecting those elements of the source on which *func* returns true.
- **reduceByKey(*func*):** when called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*.
- **sortByKey([*ascending*],[*numTasks*]):** when called on a dataset of (K, V) pairs where K implements **Ordered**, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the Boolean ***ascending*** argument.
- **join(*otherDataset*, [*numTasks*]):** when called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key.
- **distinct([*numTasks*]):** returns a new dataset that contains the distinct elements of the source dataset.
- **pipe(*command*, [*envVars*]):** pipes each partition of the RDD through the provided shell *command*

Example: Transformations

```
neg_values = dist_array.map(lambda x : -1 * x)
neg_values.collect()
```

```
[-1,
 -2,
 -3,
 -4,
 -5,
 -6,
 -7,
 -8,
 -9,
 -10,
 -11,
 -12,
 -13,
 -14,
 -15,
 -16,
 -17,
 -18,
 -19]
```

```
large_values = dist_array.filter(lambda y: y > 10)
large_values.collect()
```

```
[11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Example Actions

- **reduce(*func*):** Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
- **foreach(*func*):** Runs the function *func* on each element in the dataset.
- **count():** returns the number of elements in the dataset.
- **first():** returns the first element in the dataset.
- **take(*n*):** returns an array with the first ***n*** elements of the dataset.
- **saveAsTextFile(*path*):** writes the elements in the dataset out to a file in HDFS (or some other file system).
- **saveAsSequenceFile(*path*):** writes the elements to HDFS in the **SequenceFile** format.

Example: Actions

```
large_values = dist_array.filter(lambda y: y > 10)
large_values.collect()
```

```
[11, 12, 13, 14, 15, 16, 17, 18, 19]
```

```
large_values.count()
```

```
9
```

```
large_values.reduce(lambda x,y : x+y)
```

```
135
```

WordCount in Spark (1 of 2)

```
constitution = sc.textFile("/user/root/constitution.txt")

wordCounts = constitution.flatMap(lambda line: line.split())
         .map(lambda word: (word, 1))
         .reduceByKey(lambda a, b: a+b)
```

```
wordCounts.take(10)
```

```
[(u'all', 37),
 (u'Jr.,', 1),
 (u'Legislatures', 3),
 (u'Roads;', 1),
 (u'bear', 2),
 (u'needful', 2),
 (u'Place', 3),
 (u'four', 2),
 (u'race,', 1),
 (u'Department', 1)]
```

WordCount in Spark (2 of 2)

```
swapped_wordCounts = wordCounts.map(lambda record : (record[1], record[0]))
sorted_counts = swapped_wordCounts.sortByKey(ascending = False)
sorted_counts.take(20)
```

```
[(662, u'the'),
 (493, u'of'),
 (293, u'shall'),
 (256, u'and'),
 (183, u'to'),
 (178, u'be'),
 (157, u'or'),
 (137, u'in'),
 (100, u'by'),
 (94, u'a'),
 (85, u'United'),
 (81, u'for'),
 (79, u'any'),
 (72, u'President'),
 (64, u'The'),
 (64, u'as'),
 (63, u'have'),
 (55, u'States,'),
 (52, u'such'),
 (47, u'State')]
```

Operations on key-value pair RDDs

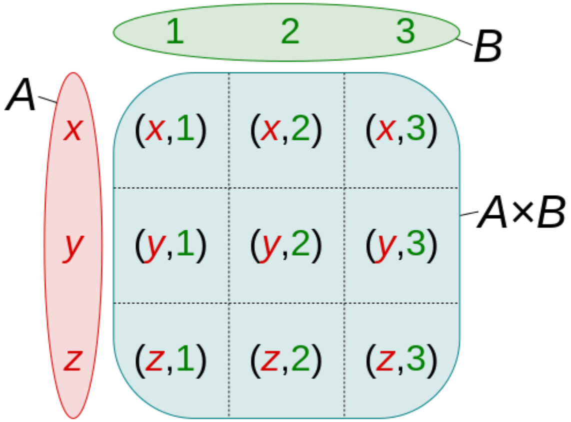
- Operations on key-value pair datasets are at the foundation of Hadoop, MapReduce, and Spark 1.6
  - **groupByKey()**: group values with the same key, ex. rdd.groupByKey()
  - **mapValues(f)**: applies function only to values not keys
  - **flatMapValues(f):** same as mapValues when function returns several values
  - **keys():** Returns RDD with only the keys
  - **values():** Returns RDD with only the values
- Simple operations on RDDs:
  - **union(otherRDD)**: makes the union of all key-value pairs

Join operations on pairs of key-value pair RDDs: join operations

- Joins are a fundamental operation of data which compute the **Cartesian product** between two sets (e.g., two RDDs)

$$A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$$

- Most of the time, joins are paired with a filter to improve performance



Join operations on pairs of key-value pair RDDs: join operations

- We can use this idea to *join* key-value pairs and then filter for pairs that have the same key
- RDD join performs an “inner join”
- Inner Join: Keys must be present in the left and right hand RDD.

```
A = sc.parallelize([
  [1,1],
  [1,2],
  [2,3]]
)
B = sc.parallelize([
  [1, "A"],
  [2, "B"]
])
```

```
A.join(B).collect()
```

```
[(1, (1, 'A')), (1, (2, 'A')), (2, (3, 'B'))]
```

Join operations on pairs of key-value pair RDDs

- Lets suppose we want to compute the total number of orders per state using:
  - Locations: locationID, state
  - Transactions: transactionID, locationID, number of orders
- Activity: how can we use join to achieve this?
- Explore leftOuterJoin and rightOuterJoin

```
locations = sc.parallelize([
  ['loc1', 'NY'],
  ['loc2', 'NY'],
  ['loc3', 'PA'],
  ['loc4', 'FL']
])
transactions = sc.parallelize([
  [1, 'loc1', 2.],
  [2, 'loc1', 3.],
  [3, 'loc2', 5.],
  [4, 'loc5', 5.]
])
```

Spark 2.0

DataFrames

DataFrames

- The problem with RDDs is that they do not have enough structure
- They are harder to optimize and therefore slow
- DataFrames tries to solve this by adding structure
- A DataFrame is a distributed collection of data organized into named columns
- Similar to Pandas DataFrames but distributed across the cluster
- You access the Spark 2.0+ functionality using the **spark** object

DataFrames (2)

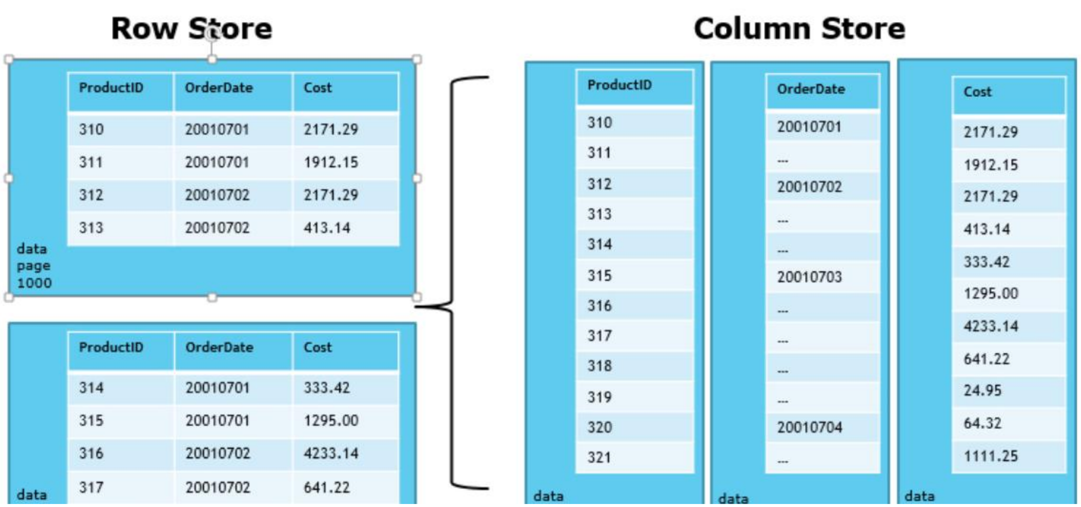
- You can read from multiple sources into dataframes

and more ...

Formats and Sources supported by DataFrames

DataFrames (3)

- One preferred source is Parquet files
- Parquet files are datasets stored in columns



DataFrame operations

- Creation of dataframes:
  - From Row objects:

```python
from pyspark.sql import Row
raw_data = [Row(state='NY', month='JAN', orders=3),
            Row(state='NJ', month='JAN', orders=4),
            Row(state='NY', month='FEB', orders=5),
           ]
data_df = spark.createDataFrame(raw_data)
```

  - From RDDs:

```python
example_rdd_with_rows.toDF()
```

  - From files:

```python
farmers_markets = spark.read.csv('/databricks-datasets/data.gov/farmers_markets_geographic_data/data-001/market_data.csv',
                                 header=True,
                                 inferSchema=True)
```

```python
spark.read.parquet(filepath)
```

DataFrame operations

- All data in a specific column has the same type
- DataFrame types can be hierarchical: The type of a column might be another “DataFrame”
- **You cannot perform dataframe operations using Python**
- Instead, you transform DataFrames by **selecting**, **modifying**, **filtering**, **joining**, **grouping**, or **aggregating** using specialized Spark commands similar to SQL
- Many of these commands are **symbolic representations** of the operations
- After the transformations, Spark builds an *execution plan* that is optimized for the column types and the operations

Exploring a DataFrame

- **printSchema():** shows the datatypes of the dataframe
- **show()**: prints the first *n* rows
- **take():** return first *n* rows
- **sample(withReplacement, fraction):** randomly sample rows (approximate)
- **display(df):** (only in DataBricks) exploratory interface for dataframe

Selecting and modifying a DataFrame

- **select(*expressions):** returns a new DataFrame with columns
- **withColumn(colName, expression):** creates a new column based on the expression
- Expressions are **symbolic operations**
- Symbolic operations can hold *literal* values and *placeholders* for column names
- You can perform symbolic operations on both literals and placeholders
- Some symbolic operations are available in pyspark.sql.functions

Symbolic operations

- expression = 1 + n_employees
- To express the previous operation, I need a *literal* value (1) and a *placeholder* for the column n_employees

```
1 + fn.col('n_employees')
```

```
t[23]: Column<(n_employees + 1)>
```

- A symbolic operation produces a *column object which contains the defined operations*

Symbolic operations

```
locations_df.select(1 + fn.col('n_employees')).show(10)
```

▸ (3) Spark Jobs

```
+-----------------+
|(n_employees + 1)|
+-----------------+
|                4|
|                9|
|                4|
|                2|
+-----------------+
```

Change column name

```
new_column = 1 + fn.col('n_employees')
locations_df.select(new_column.alias('new_column')).show(10)
```

▸ (3) Spark Jobs

```
+----------+
|new_column|
+----------+
|         4|
|         9|
|         4|
|         2|
+----------+
```

Change column type

```
locations_df.\
  select(new_column.alias('new_column').cast('float')).\
  show(10)
```

▸ (3) Spark Jobs

```
+----------+
|new_column|
+----------+
|       4.0|
|       9.0|
|       4.0|
|       2.0|
+----------+
```

Selecting and modifying (1)

- You can use select to select columns, modify them, or create new columns

```
locations_df.show(10)
```

▸ (3) Spark Jobs

```
+-----------+-----------+-----+
|location_id|n_employees|state|
+-----------+-----------+-----+
|       loc1|          3|   NY|
|       loc2|          8|   NY|
|       loc3|          3|   PA|
|       loc4|          1|   FL|
+-----------+-----------+-----+
```

```
locations_df.\
  select('n_employees',
         'location_id',
         'state',
         (fn.col('n_employees') + 1).alias('n_employees_plus_1'),
         (fn.col('n_employees') > 5).alias('more_than_5_empl')
        ).\
  show(10)
```

▸ (3) Spark Jobs

```
+-----------+-----------+-----+------------------+----------------+
|n_employees|location_id|state|n_employees_plus_1|more_than_5_empl|
+-----------+-----------+-----+------------------+----------------+
|          3|       loc1|   NY|                 4|           false|
|          8|       loc2|   NY|                 9|            true|
|          3|       loc3|   PA|                 4|           false|
|          1|       loc4|   FL|                 2|           false|
+-----------+-----------+-----+------------------+----------------+
```

Selecting and modifying (2)

- The following code snippet in that new columns are created using utility functions provided by the Spark SQL utility function (fn) module.
- Note that fn.lit() is needed to create a literal passed to pow

```
locations_df.\
  select(fn.sqrt('n_employees'),
         'n_employees',
         fn.pow(fn.col('n_employees'),
                fn.lit(2))).\
  show()
```

▸ (3) Spark Jobs

```
+-----------------+-----------+--------------------+
| SQRT(n_employees)|n_employees|POWER(n_employees, 2)|
+-----------------+-----------+--------------------+
|1.7320508075688772|          3|                 9.0|
|2.8284271247461903|          8|                64.0|
|1.7320508075688772|          3|                 9.0|
|               1.0|          1|                 1.0|
+-----------------+-----------+--------------------+
```

Filtering

- **where(expression)**: select only rows where expression is true
- Expressions can be complex:

```
locations_df.where((fn.col('n_employees') > 2) & (fn.col('state') == 'PA')).show()
```

▸ (3) Spark Jobs

```
+-----------+-----------+-----+
|location_id|n_employees|state|
+-----------+-----------+-----+
|       loc3|          3|   PA|
+-----------+-----------+-----+
```

Joining

- **Inner joins** (keep rows with keys that exist in the left and right datasets)
- **Outer joins** (keep rows with keys in either the left or right datasets)
- **Left outer joins** (keep rows with keys in the left dataset)
- **Right outer joins** (keep rows with keys in the right dataset)
- Left semi joins (keep the rows in the left, and only the left, dataset where the key appears in the right dataset)
- Left anti joins (keep the rows in the left, and only the left, dataset where they do not appear in the right dataset)
- Natural joins (perform a join by implicitly matching the columns between the two datasets with the same names)
- Cross (or Cartesian) joins (match every row in the left dataset with every row in the right dataset)

Inner Join

- Keep rows with keys that exist in left and right data frames

```
1  locations_df.join(transactions_df, on='location_id').show()
```

▸ (5) Spark Jobs

```
+-----------+-----------+-----+--------+--------------+
|location_id|n_employees|state|n_orders|transaction_id|
+-----------+-----------+-----+--------+--------------+
|       loc1|          3|   NY|     2.0|             1|
|       loc1|          3|   NY|     3.0|             2|
|       loc3|          3|   PA|     5.0|             3|
+-----------+-----------+-----+--------+--------------+
```

```
1  locations_df.show(10)
```

▸ (3) Spark Jobs

```
+-----------+-----------+-----+
|location_id|n_employees|state|
+-----------+-----------+-----+
|       loc1|          3|   NY|
|       loc2|          8|   NY|
|       loc3|          3|   PA|
|       loc4|          1|   FL|
+-----------+-----------+-----+
```

```
1  transactions_df.show(10)
```

▸ (3) Spark Jobs

```
+-----------+--------+--------------+
|location_id|n_orders|transaction_id|
+-----------+--------+--------------+
|       loc1|     2.0|             1|
|       loc1|     3.0|             2|
|       loc3|     5.0|             3|
|       loc5|     5.0|             4|
+-----------+--------+--------------+
```

Left outer join

- Keep rows with keys that exist in left data frame

```
1  locations_df.join(transactions_df, on='location_id', how='left').show()
```

▸ (5) Spark Jobs

```
+-----------+-----------+-----+--------+--------------+
|location_id|n_employees|state|n_orders|transaction_id|
+-----------+-----------+-----+--------+--------------+
|       loc1|          3|   NY|     2.0|             1|
|       loc1|          3|   NY|     3.0|             2|
|       loc3|          3|   PA|     5.0|             3|
|       loc2|          8|   NY|    null|          null|
|       loc4|          1|   FL|    null|          null|
+-----------+-----------+-----+--------+--------------+
```

```
1  locations_df.show(10)
```

▸ (3) Spark Jobs

```
+-----------+-----------+-----+
|location_id|n_employees|state|
+-----------+-----------+-----+
|       loc1|          3|   NY|
|       loc2|          8|   NY|
|       loc3|          3|   PA|
|       loc4|          1|   FL|
+-----------+-----------+-----+
```

```
1  transactions_df.show(10)
```

▸ (3) Spark Jobs

```
+-----------+--------+--------------+
|location_id|n_orders|transaction_id|
+-----------+--------+--------------+
|       loc1|     2.0|             1|
|       loc1|     3.0|             2|
|       loc3|     5.0|             3|
|       loc5|     5.0|             4|
+-----------+--------+--------------+
```

Right outer join

- Keep rows with keys that exist in right data frame

```
1  locations_df.join(transactions_df, on='location_id', how='right').show()
```

▸ (5) Spark Jobs

```
+-----------+-----------+-----+--------+--------------+
|location_id|n_employees|state|n_orders|transaction_id|
+-----------+-----------+-----+--------+--------------+
|       loc1|          3|   NY|     2.0|             1|
|       loc1|          3|   NY|     3.0|             2|
|       loc3|          3|   PA|     5.0|             3|
|       loc5|       null| null|     5.0|             4|
+-----------+-----------+-----+--------+--------------+
```

```
1  locations_df.show(10)
```

▸ (3) Spark Jobs

```
+-----------+-----------+-----+
|location_id|n_employees|state|
+-----------+-----------+-----+
|       loc1|          3|   NY|
|       loc2|          8|   NY|
|       loc3|          3|   PA|
|       loc4|          1|   FL|
+-----------+-----------+-----+
```

```
1  transactions_df.show(10)
```

▸ (3) Spark Jobs

```
+-----------+--------+--------------+
|location_id|n_orders|transaction_id|
+-----------+--------+--------------+
|       loc1|     2.0|             1|
|       loc1|     3.0|             2|
|       loc3|     5.0|             3|
|       loc5|     5.0|             4|
+-----------+--------+--------------+
```

Outer join

- Keep rows with keys that exist in either the left or right data frames

```
locations_df.join(transactions_df, on='location_id', how='outer').show()
```

(5) Spark Jobs

```
+-----------+-----------+-----+--------+--------------+
|location_id|n_employees|state|n_orders|transaction_id|
+-----------+-----------+-----+--------+--------------+
|       loc1|          3|   NY|     2.0|             1|
|       loc1|          3|   NY|     3.0|             2|
|       loc3|          3|   PA|     5.0|             3|
|       loc2|          8|   NY|    null|          null|
|       loc4|          1|   FL|    null|          null|
|       loc5|       null| null|     5.0|             4|
+-----------+-----------+-----+--------+--------------+
```

```
locations_df.show(10)
```

(3) Spark Jobs

```
+-----------+-----------+-----+
|location_id|n_employees|state|
+-----------+-----------+-----+
|       loc1|          3|   NY|
|       loc2|          8|   NY|
|       loc3|          3|   PA|
|       loc4|          1|   FL|
+-----------+-----------+-----+
```

```
transactions_df.show(10)
```

(3) Spark Jobs

```
+-----------+--------+--------------+
|location_id|n_orders|transaction_id|
+-----------+--------+--------------+
|       loc1|     2.0|             1|
|       loc1|     3.0|             2|
|       loc3|     5.0|             3|
|       loc5|     5.0|             4|
+-----------+--------+--------------+
```

Grouping

- **groupBy(*expressions)**: groups by a list of expressions
- Typically, a list of columns:

```
locations_df.\
  join(transactions_df, on='location_id').\
  groupBy('state')
```

```
Out[76]: <pyspark.sql.group.GroupedData at 0x7f4ac607ee50>
```

- Note that this transformation not do anything until we perform an action on the group object like an aggregate action.

Aggregate

- There are some special functions that only work on grouped data
- They are applied using the method **agg(*expressions)**
- For example:
    - fn.sum, fn.stddev: self explanatory
    - fn.count: counts when column is not null
    - fn.countDistinct: how many distinct values of a column

Aggregate

```
locations_df.\
  join(transactions_df, on ='location_id').\
  groupBy('state').\
  agg(fn.sum('n_orders')).\
  show()
```

▸ (5) Spark Jobs

```
+-----+-------------+
|state|sum(n_orders)|
+-----+-------------+
|   PA|          5.0|
|   NY|          5.0|
+-----+-------------+
```

Spark ML

- Spark ML implements several machine learning algorithms at scale:
  - Regression
  - Classification
  - Decision Trees
  - Clustering
- It works on Spark DataFrames by performing transformations of the data
- A typical data science analysis requires several transformations
- These transformations can be implemented through Pipelines

Spark ML (2)

- A model is known as **Estimator** in Spark ML and the typical cycle for such objects is as follows
  1. Define zero or more **input columns**
  2. Define zero or more **output columns**
  3. Define **parameters** of the estimator
  4. **Fit** the estimator, which returns a **fitted model**
  5. Use the fitted model to perform **transformations**

Example Spark machine learning workflow

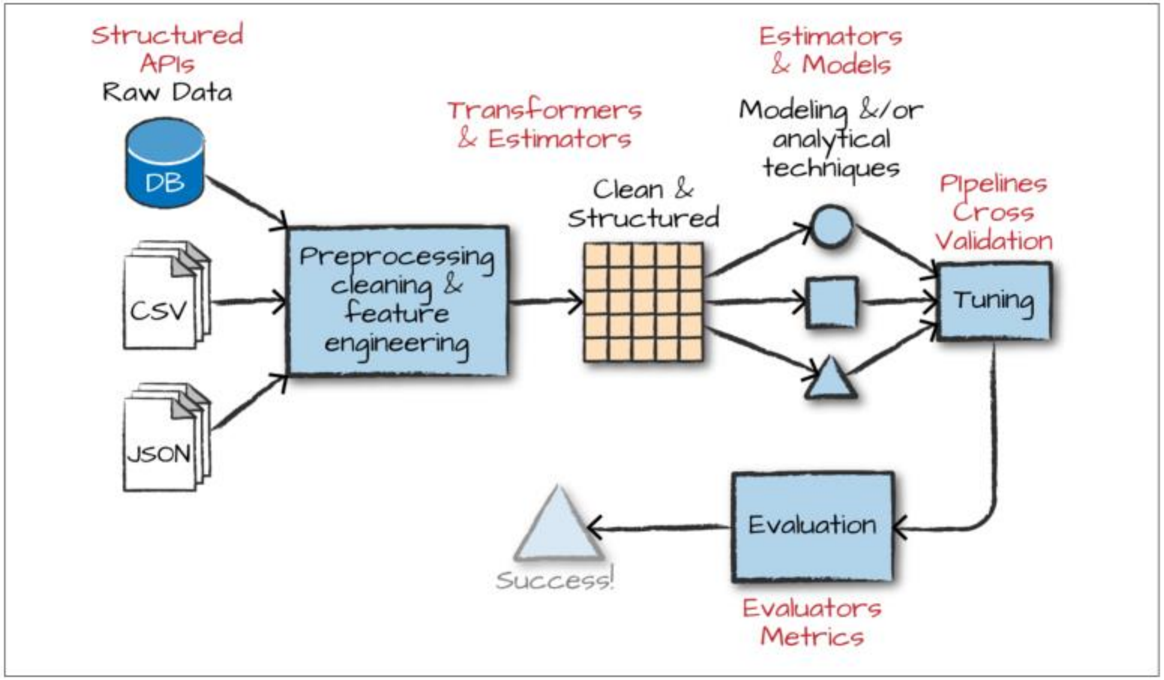
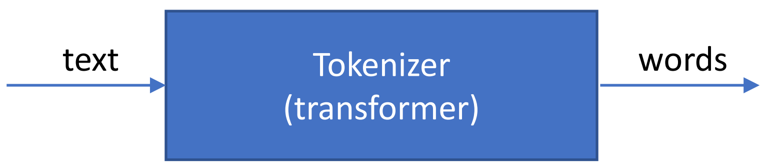


Figure 24-2. The machine learning workflow, in Spark

Spark ML: Transformers and Estimators

- Note that Spark ML for RDD is getting deprecated
- Preferred method in the future is to work only with DataFrames.
- Spark Machine Learning works by creating transformers and estimators DataFrames as inputs

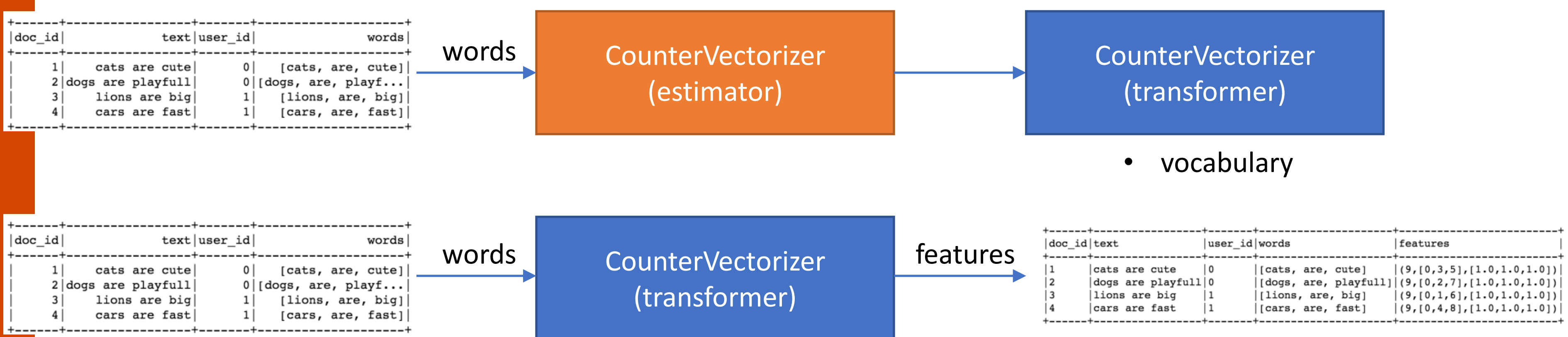
doc_id	text	user_id
1	cats are cute	0
2	dogs are playfull	0
3	lions are big	1
4	cars are fast	1



doc_id	text	user_id	words
1	cats are cute	0	[cats, are, cute]
2	dogs are playfull	0	[dogs, are, playf...
3	lions are big	1	[lions, are, big]
4	cars are fast	1	[cars, are, fast]

Spark ML: Transformers and Estimators

- **Transformers** convert raw data in some way
- **Estimators** need to learn something from the data in order to transform the data
- For example, if we want to count terms in text, we need to know how many terms are in all documents



doc_id	text	user_id	words
1	cats are cute	0	[cats, are, cute]
2	dogs are playfull	0	[dogs, are, playf...
3	lions are big	1	[lions, are, big]
4	cars are fast	1	[cars, are, fast]

doc_id	text	user_id	words
1	cats are cute	0	[cats, are, cute]
2	dogs are playfull	0	[dogs, are, playf...
3	lions are big	1	[lions, are, big]
4	cars are fast	1	[cars, are, fast]

doc_id	text	user_id	words	features
1	cats are cute	0	[cats, are, cute]	(9,[0,3,5],[1.0,1.0,1.0])
2	dogs are playfull	0	[dogs, are, playfull]	(9,[0,2,7],[1.0,1.0,1.0])
3	lions are big	1	[lions, are, big]	(9,[0,1,6],[1.0,1.0,1.0])
4	cars are fast	1	[cars, are, fast]	(9,[0,4,8],[1.0,1.0,1.0])

Example Spark Transformer

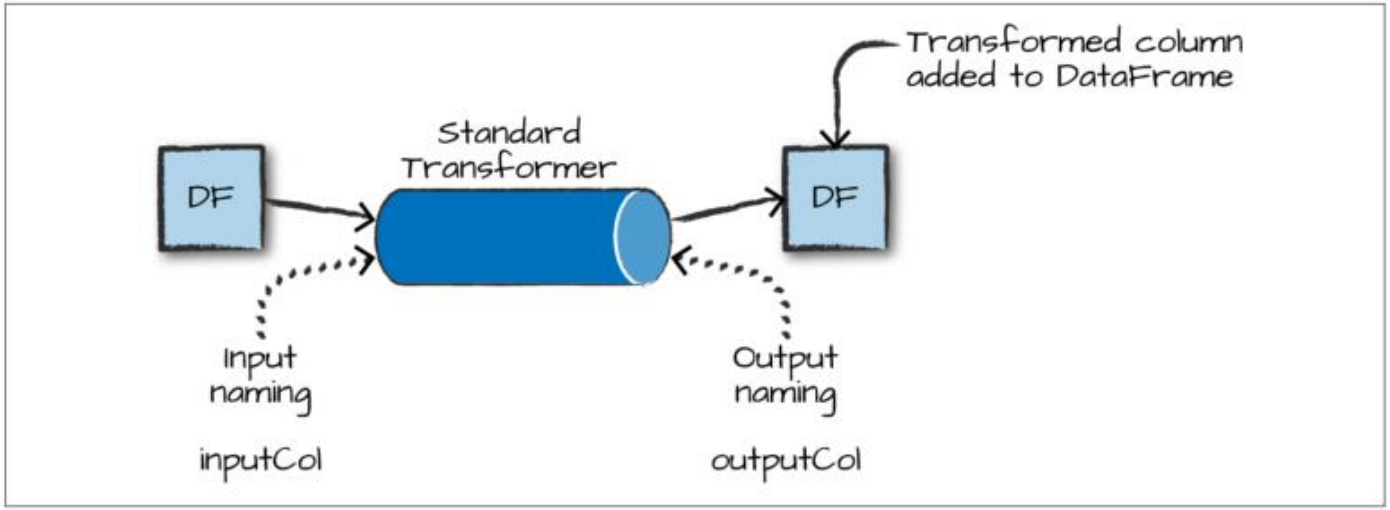
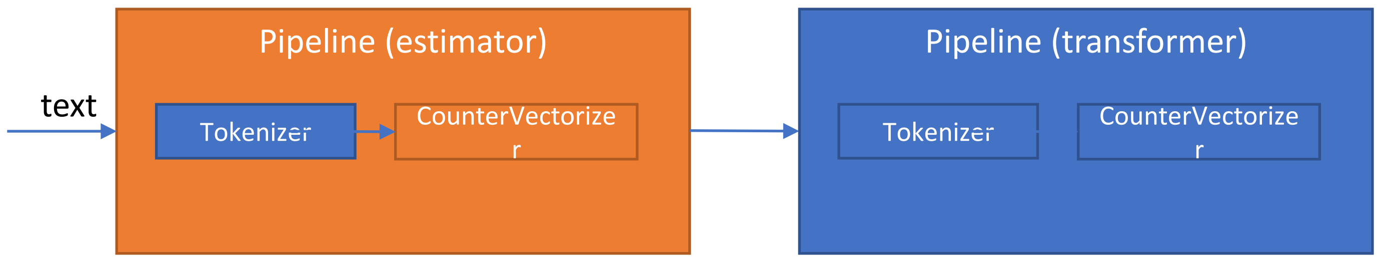


Figure 24-3. A standard transformer

Spark ML: Pipelines

- Data science is all about building data analytic pipelines from raw data to models
- Spark ML Pipeline chains multiple Transformers and Estimators
- Pipelines can be saved and shared
- A Pipeline always starts as an Estimator that needs to be fitted

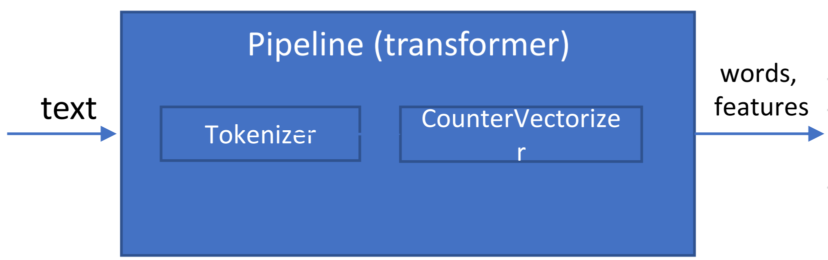
doc_id	text	user_id
1	cats are cute	0
2	dogs are playfull	0
3	lions are big	1
4	cars are fast	1



Spark ML: Pipelines

- A Pipeline **always** needs to be fitted even when stages are all transformers
- A Pipeline transformer contains an entire process

doc_id	text	user_id
1	cats are cute	0
2	dogs are playfull	0
3	lions are big	1
4	cars are fast	1



doc_id	text	user_id	words	features
1	cats are cute	0	[cats, are, cute]	(9,[0,3,5],[1.0,1.0,1.0])
2	dogs are playfull	0	[dogs, are, playfull]	(9,[0,2,7],[1.0,1.0,1.0])
3	lions are big	1	[lions, are, big]	(9,[0,1,6],[1.0,1.0,1.0])
4	cars are fast	1	[cars, are, fast]	(9,[0,4,8],[1.0,1.0,1.0])

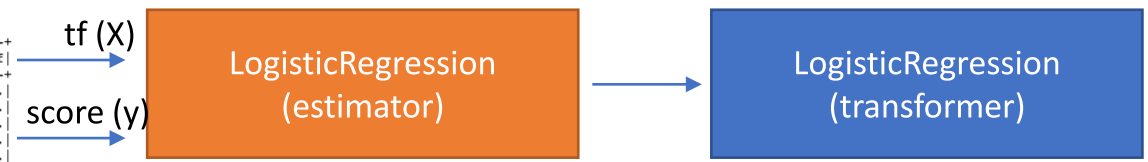
Spark ML: Algorithms

- Some Estimators are algorithms that work on DataFrames
- In this context, an algorithm is a machine learning model
- For example, LogisticRegression

Training data

id	review	score	words	tf
neg_1	Robert DeNiro pla...	0.0	[robert, deniro, ...	(26677,[0,1,2,4,8...
neg_10023	Shame on Yash Raj...	0.0	[shame, on, yash,...	(26677,[3,4,6,9,1...
neg_10032	The storyline was...	0.0	[the, storyline, ...	(26677,[0,1,2,3,4...
neg_10049	I love the freque...	0.0	[i, love, the, fr...	(26677,[0,4,5,10,...
neg_10140	This movie should...	0.0	[this, movie, sho...	(26677,[0,2,8,11,...

only showing top 5 rows



Testing data

id	review	score	words	tf
neg_10023	Shame on Yash Raj...	0.0	[shame, on, yash,...	(26677,[3,4,6,9,1...
neg_10032	The storyline was...	0.0	[the, storyline, ...	(26677,[0,1,2,3,4...
neg_10049	I love the freque...	0.0	[i, love, the, fr...	(26677,[0,4,5,10,...
neg_10140	This movie should...	0.0	[this, movie, sho...	(26677,[0,2,8,11,...
neg_102	My girlfriend onc...	0.0	[my, girlfriend, ...	(26677,[0,3,4,6,2...

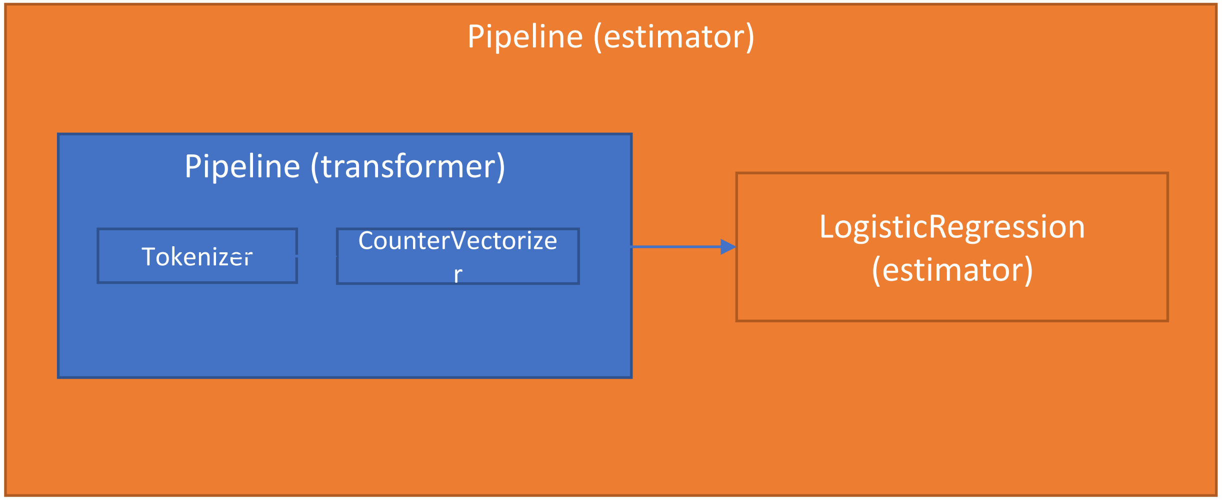
only showing top 5 rows

id	review	score	words	tf	prediction
neg_10023	Shame on Yash Raj...	0.0	[shame, on, yash,...	(26677,[3,4,6,9,1...	0.0
neg_10032	The storyline was...	0.0	[the, storyline, ...	(26677,[0,1,2,3,4...	0.0
neg_10049	I love the freque...	0.0	[i, love, the, fr...	(26677,[0,4,5,10,...	1.0
neg_10140	This movie should...	0.0	[this, movie, sho...	(26677,[0,2,8,11,...	0.0
neg_102	My girlfriend onc...	0.0	[my, girlfriend, ...	(26677,[0,3,4,6,2...	0.0

only showing top 5 rows

Spark ML: Algorithms

- Algorithms can be part of Pipelines
- Pipelines can be combined with other Pipelines



Evaluators

- An evaluator allows us to see how a given model performs according to specified criteria
- An evaluator is used to select the best model from a group of models.
- An evaluator compares model predictions to truth data and calculates a score indicating how well the model performed.

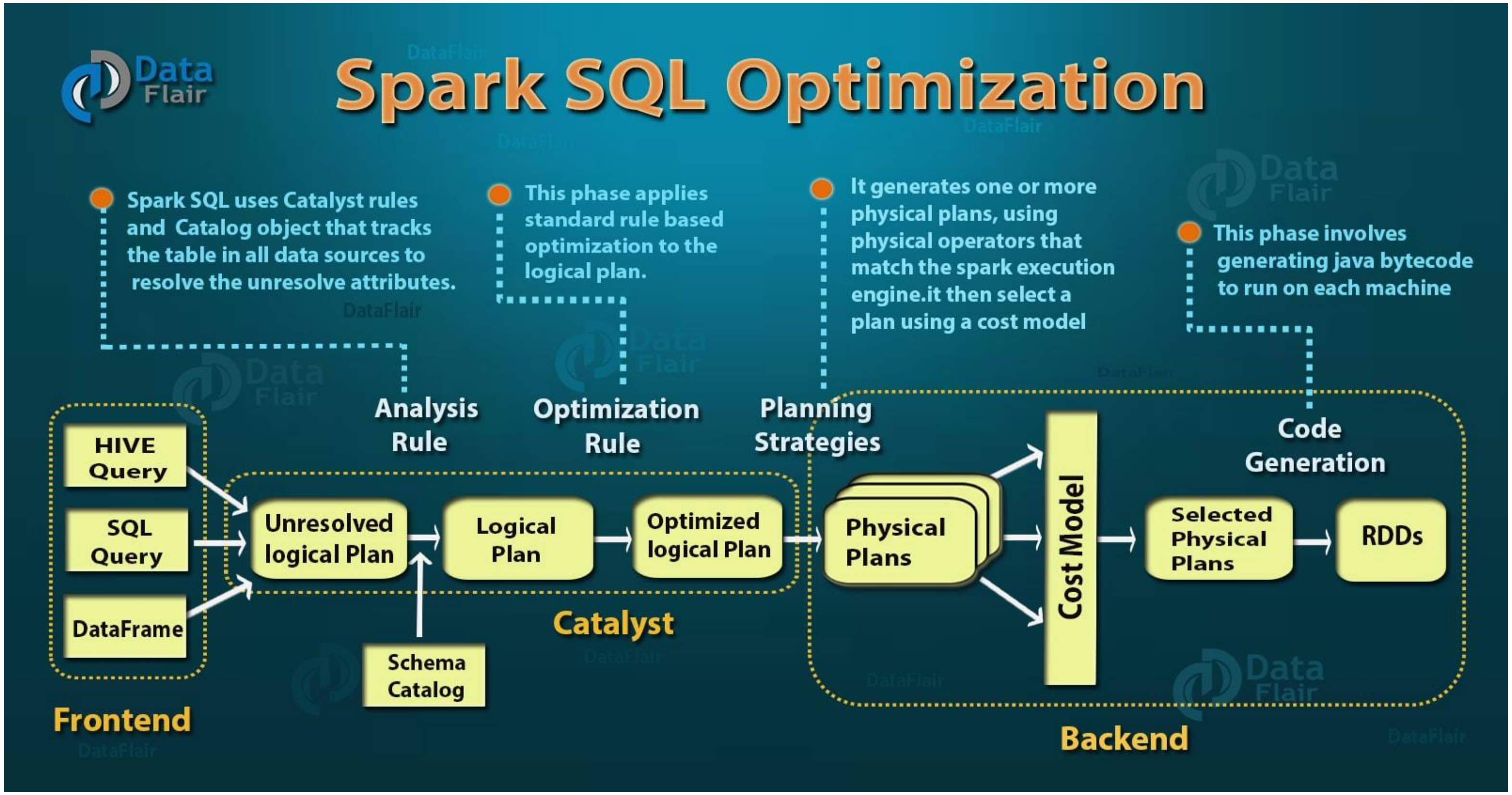
Summary (1 of 2)

- ***Apache Spark*** is a fast, in-memory computing system that runs on Hadoop.
- An application in Spark has several main components, including a driver program, worker nodes, executors and tasks.
- The **SparkContext** object is a key component of the driver program.
- An RDD is a fault-tolerant collection of elements that can be operated on in parallel.
- There are several ways to create an RDD in Spark: the **parallelize** function or a reference to a file in HDFS. **textFile sequenceFile**
- There are two types of operations that can be done on RDDs: ***transformations*** and ***actions***.

Summary (2 of 2)

- ***MLlib*** is a Spark implementation of some common machine learning algorithms and utilities.
- MLlib contains **algorithms** for classification, regression, clustering and recommendation.
- **Transformers** convert raw data in some way
- **Estimators** need to learn something from the data in order to transform the data
- **Pipelines** are a collection of transformers and estimators
- Algorithms are

Spark Sql



Data Flair
Spark SQL Optimization
Spark SQL uses Catalyst rules and Catalog object that tracks the table in all data sources to resolve the unresolve attributes.
This phase applies standard rule based optimization to the logical plan.
It generates one or more physical plans, using physical operators that match the spark execution engine.it then select a plan using a cost model
This phase involves generating java bytecode to run on each machine
Analysis Rule
Optimization Rule
Planning Strategies
Code Generation
HIVE Query
SQL Query
DataFrame
Unresolved logical Plan
Logical Plan
Optimized logical Plan
Physical Plans
Cost Model
Selected Physical Plans
RDDs
Schema Catalog
Catalyst
Frontend
Backend

03 Spark and Hadoop

Hadoop is a **framework** for distributed processing and is comprised of YARN, MapReduce, HDFS and common modules.

Hadoop **ecosystem** adds to its capabilities: HBase, Storm, Mahout, Neo4j, Graph, etc.

With capabilities in file distribution, data security, resource management and DR, Hadoop is **a de facto standard** for Big Data.

MapReduce processing is **disk-based**, scaling by adding cheap computers and cheap disk.

Largest known cluster: Yahoo 42,000 nodes. Hadoop MapReduce uses **batch processing** and built ideally to crawl through web sites.

Spark is a **general-purpose** data processing engine that can run on its own or in Hadoop clusters through YARN.

Spark **capabilities** include SQL, streaming, machine learning and graph processing.

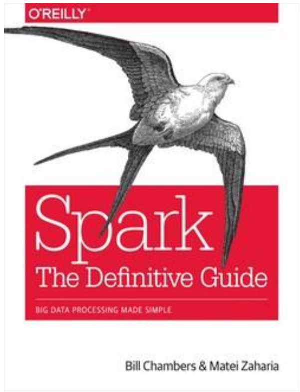
Spark is not a replacement for Hadoop but can provide an **alternative to Hadoop MapReduce** under several scenarios.

Spark provides rapid **in-memory** processing and scaling comes at a cost (memory).

Largest know cluster: 8,000 nodes. Spark excels at **streaming** workloads, interactive queries, and machine-based learning.

Note: Hadoop and Spark can co-exist and is often the recommended solution for new implementations.

Reference books



Reference LINKS

https://www.oreilly.com/library/view/spark-the-definitive/9781491912201/

https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

https://data-flair.training/blogs/apache-spark-rdd-vs-dataframe-vs-dataset/

http://www.jamesserra.com/archive/2017/12/is-the-traditional-data-warehouse-dead/